A Programming Language for Distributed Systems

Pascal Weisenburger

A Programming Language for Distributed Systems

vom Fachbereich Informatik
der Technischen Universität Darmstadt

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte

Dissertation von
Pascal Weisenburger

Referent Prof. Dr. Guido Salvaneschi

Korreferent Prof. Dr. Philipp Haller

Darmstadt, 2020

Pascal Weisenburger: A Programming Language for Distributed Systems
Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2020
URN: urn:nbn:de:tuda-tuprints-135006
Tag der Prüfung: 21.08.2020
Darmstadt – D 17

Veröffentlicht unter CC BY-SA 4.0 International
https://creativecommons.org/licenses/by-sa/4.0/

Acknowledgements

My thanks especially go to my advisor, Guido Salvaneschi, for providing me with the opportunity to work on the language presented in this thesis, giving me the freedom to explore its design and helping in shaping its ideas.

I thank Mira Mezini for her support and for inspiring my interest in programming languages. I also thank Philipp Haller for being the second examiner of my thesis and his time and effort of carefully reviewing it. Furthermore, I would like to thank the remaining members of my PhD committee: Sebastian Faust, Matthias Hollick and Kirstin Peters. My special thanks go to Gudrun Harris for managing the group so smoothly, for keeping us from having to be concerned about our contracts and for saving us from most of the paperwork. I also thank my long-time office colleagues Matthias Eichholz, Mirko Köhler and Ragnar Mogk for the fruitful collaboration and the nice office atmosphere and the members Nafise Eskandani, Aditya Oak, Simon Schönwälder and Daniel Sokolowski of the RP group and the members of the STG group.

I gratefully thank my family for their encouragement and continued support in every imaginable way.

To everyone who accompanied me in the last years on this academic journey:
Thank you!
That was fun.

Abstract

Today's software, including many everyday services, such as online streaming, search engines and social networks, is widely distributed, running on top of a network of interconnected computers. Such distributed applications are traditionally developed as separate modules for each component in the distributed system. These modules react to events, like user input or messages from the network, and in turn produce new events for the other modules. Separation into different modules is problematic because combining modules is notoriously hard and requires extensive and time-consuming integration and manual implementation of communication forces programmers to program complex event-based communication schemes among hosts – an activity which is often low-level and error-prone. The combination of the two results in obscure distributed data flows scattered among multiple modules, hindering reasoning about the system as a whole. For these reasons, despite most software today is distributed, the design and development of distributed systems remains surprisingly challenging.

We present the ScalaLoci distributed programming language, our approach for taming the complexity of developing distributed applications via specialized programming language support. ScalaLoci addresses the issues above with a coherent model based on placement types that enables reasoning about distributed data flows otherwise scattered across multiple modules, supporting multiple software architectures via dedicated language features and abstracting over low-level communication details and data conversions.

ScalaLoci does not force developers to modularize software along network boundaries as is traditionally the case when developing distributed systems. Instead, we propose a module system that supports encapsulating each (cross-host) functionality and defining it over abstract peer types. As a result, we disentangle modularization and distribution and we enable the definition of a distributed system as a composition of ScalaLoci modules, each representing a subsystem.

Our case studies on distributed algorithms, distributed data structures, as well as on real-world distributed streaming engines show that ScalaLoci simplifies developing distributed systems, reduces error-prone communication code and favors early detection of bugs. As we demonstrate, the ScalaLoci module system allows the definition of reusable patterns of interaction in distributed software and enables separating the modularization and distribution concerns, properly separating functionalities in distributed systems.

Zusammenfassung

Heutige Software, einschließlich vieler alltäglicher Dienste wie Online-Streaming, Suchmaschinen und soziale Netzwerke, wird auf einem Netzwerk miteinander verbundener Computer verteilt ausgeführt. Solche verteilten Anwendungen werden traditionell mit separaten Modulen für jede Komponente im verteilten System entwickelt. Module reagieren auf Ereignisse wie Benutzereingaben oder Netzwerknachrichten und produzieren wiederum neue Ereignisse für die anderen Module. Die Trennung in verschiedene Module ist problematisch, weil das Kombinieren von Modulen notorisch schwierig ist und umfangreiche und zeitraubende Integration erfordert und die manuelle Implementierung der Kommunikation die Programmierer dazu zwingt, komplexe ereignisbasierte Kommunikationsschemata zwischen Hostrechnern zu implementieren – was oft auf niedriger Abstraktionsebene stattfindet und fehleranfällig ist. Die Kombination von beidem führt zu unübersichtlichen verteilten Datenströmen, die über mehrere Module hinweg aufgesplittet sind und das Verständnis über das System als Ganzes erschweren. Aus diesen Gründen bleibt der Entwurf und die Entwicklung verteilter Systeme trotz der Tatsache, dass die meiste Software heutzutage verteilt ist, überraschend schwierig.

Diese Dissertation präsentiert die Programmiersprache ScalaLoci für die Entwicklung verteilter Anwendungen. Der vorgestellte Ansatz basiert auf spezialisierter Programmiersprachenunterstützung, um die Komplexität bei der Entwicklung verteilter Anwendungen zu bewältigen. ScalaLoci geht die oben genannten Probleme mit einem kohärenten Modell basierend auf Platzierungstypen an, das die Betrachtung verteilter Datenflüsse ermöglicht, die ansonsten über mehrere Module aufgesplittet sind. ScalaLoci unterstützt dabei mehrere Softwarearchitekturen durch dedizierte Sprachmerkmale und abstrahiert über Kommunikationsdetails und Datenkonvertierungen.

ScalaLoci zwingt Entwickler nicht dazu, Software entlang von Netzwerkgrenzen zu modularisieren, wie es traditionell bei der Entwicklung verteilter Systeme der Fall ist. Stattdessen stellt ScalaLoci ein Modulsystem bereit, das die Kapselung jeder (Host-übergreifenden) Funktionalität und deren Definition über abstrakte Peer-Typen unterstützt. Als Ergebnis werden Modularisierung- und Verteilungsbelange entflochten und die Definition eines verteilten Systems als eine Zusammensetzung von ScalaLoci-Modulen, die jeweils ein Subsystem repräsentieren, ermöglicht.

Fallstudien zu verteilten Algorithmen, verteilten Datenstrukturen sowie zu verteilten Streaming-Engines aus der Praxis zeigen, dass ScalaLoci die Entwicklung

verteilter Systeme vereinfacht, fehleranfälligen Kommunikations-Code reduziert und das frühzeitige Erkennen von Fehlern unterstützt. Die Evaluation zeigt weiterhin, dass das ScalaLoci-Modulsystem die Definition wiederverwendbarer Interaktionsmuster in verteilter Software erlaubt und die Trennung von Modularisierung und Verteilung ermöglicht, indem Funktionalitäten in verteilten Systemen sauber voneinander getrennt werden.

Contents

1 Introduction

> *Things are only impossible until they're not.*
>
> **— Jean-Luc Picard**

Modern software is often distributed, building on top of networked computers, which are usually geographically spread and communicate via wired or wireless connections [Steen and Tanenbaum 2016]. A variety of distributed applications, whose size can range from just a few devices to more than thousands, have emerged in today's widely distributed and heterogeneous environments of the Web, the cloud, mobile platforms and the Internet of Things. Many day-to-day services, such as online streaming, search engines and social networks, are powered by distributed systems. Such systems need to integrate code across mobile devices, personal computers and cloud services [Zhang et al. 2014] and require developers to understand distributed program execution as an inherent aspect of the system.

Implementing distributed systems, however, is notoriously difficult because of a number of issues that naturally arise in this setting, such as consistency, concurrency, availability through data replication, fault tolerance, mismatch among data formats, as well as mix of languages and execution platforms. Those issues add significant complexity over programming local applications and are not specifically supported by existing programming languages. More generally, widely used languages provide poor abstractions for distribution, either (i) making distribution completely transparent, e.g., distributed shared memory [Nitzberg and Lo 1991], which hides significant run time differences between local and remote access concerning latency and potential network failure, hindering the development of performant and fault-tolerant code, (ii) providing a high-level programming model only for a specific domain, such as big data or streaming systems, e.g., MapReduce [Dean and Ghemawat 2008], Spark's Discretized Streams [Zaharia et al. 2012], Flink [Carbone et al. 2015], or (iii) being general purpose but providing only low-level communication primitives, such as message passing in the actor model [Agha 1986].

Despite a long history of research, developing distributed applications remains challenging. Among the sources of complexity, we find that distributed applications require transferring both data and control among different hosts [Thekkath et al. 1994] and are often event-based [Carzaniga et al. 2001; Meier and Cahill 2002].

These two aspects complicate reasoning about the distributed system because its run time behavior depends on the interaction among separate modules via events, whose occurrences can be unpredictable and potentially interleaving [Edwards 2009; Fischer et al. 2007]. First, separate development of each module limits programmers to only a view on the local component, which hinders understanding the interactions throughout the entire distributed system. Second, keeping track of potential events, control flows and data flows among components may become cumbersome.

Relevant ideas to tackle those issues emerged in the field of *multitier* – or *tierless* – programming, which investigates effective techniques to abstract over code separation into different distributed components, lowering the effort of developing distributed applications.

Multitier languages aim at bringing the development of distributed systems closer to programming single-host applications by providing means to abstract over distribution and remote communication among different components. With multitier languages, programmers use a *single language* and mix functionalities that belong to different *tiers*, e.g., the client and the server, *inside the same compilation unit*. The compiler automatically splits such compilation unit into the modules to deploy on each tier, performs the necessary translations (e.g., translating client code to JavaScript) and generates the necessary communication code [Cooper et al. 2006; Neubauer and Thiemann 2005; Serrano et al. 2006]. As a result, developers use a single language, are not forced to worry about network communication, serialization, data formats and conversions and can focus on the application logic without breaking it down along network boundaries.

Multitier programming is a highly promising direction for helping developing distributed systems. Hence, our language design that targets generic distributed systems borrows ideas from the multitier approach for specifying distribution. For declaratively defining data flows across components, we draw on *reactive programming*, which is a technique that allows direct data flow specification. Yet, although software today is often distributed with data flows spanning across multiple hosts, today's language abstractions for developing distributed systems are insufficient to tackle such complexity.

1.1 Problem

Developing general distributed systems is hard and lacks proper language support. While the techniques above provide essential features and present great potential for reducing the complexity of developing distributed systems, they do not achieve

their full potential for supporting the development of distributed systems. Our key insight is that a novel language abstraction that we propose – *placement types* – to associate locations to data and to computations enables the simplification of distributed system development by implementing distributed systems in multitier style and defining the communication among components in a declarative manner.

Thus, we claim that the complexity of developing a distributed application is in a major fraction accidental and due to poor abstractions and can be significantly addressed with new language abstractions that consider the following problem areas.

Lack of generality Distributed systems exhibit a variety of distributed architectures. We take the core idea from multitier languages of language-level support for specifying the location where code is executed. Existing multitier languages, however, target the client–server model, mostly in web applications, lacking support for *generic* distributed architectures. These languages [Chlipala 2015; Cooper et al. 2006; Philips et al. 2018; Radanne et al. 2016; Rajchenbach-Teller and Sinot 2010; Reynders et al. 2020; Serrano et al. 2006] focus on issues like the impedance mismatch between server-side code and JavaScript, integrating database queries and manipulating the DOM. Also, the Web setting allows for certain assumptions, like client-driven interaction or stateless REST servers, which do not hold for generic distributed systems.

Lack of data flow abstractions Distributed applications are in many cases reactive [Eugster et al. 2003; Pietzuch and Bacon 2002]. Events, e.g., network messages or user input, transfer data among hosts and trigger state changes or new events in the system. Hence, reactive programming, which focuses on value propagation through data flows, is a natural fit for implementing the communication in distributed systems. Yet, existing reactive programming abstractions do not cross the boundaries of components and existing multitier languages (e.g., Hop [Serrano et al. 2006] or Links [Cooper et al. 2006]) do not support reactive programming. Multitier languages that provide reactive features (e.g., Gavial [Reynders et al. 2020]) are limited to the Web domain. Some multitier languages that feature reactive abstractions (e.g., Ur/Web [Chlipala 2015]) further confine reactive values to a single component and do not allow defining data flows over multiple hosts.

Lack of modularization abstractions Complex distributed applications need to be properly modularized. Traditionally, modularization follows network boundaries due to the lack of language abstractions for defining distributed functionality in a single module. The multitier approach lifts such restriction, allowing program-

mers to bundle the code that belongs to a single (distributed) functionality inside the same module. Scaling multitier code to large applications, however, is an open problem. Researchers have been focusing on small use cases that primarily aim to demonstrate the design of their language rather than investigating the development of complex applications that require sophisticated modularization and composition. Current multitier languages do not support dedicated modularization abstractions for programming in the large such as module systems [Leroy 2000]. In its current state, multitier programming removes the need to modularize applications along network boundaries. Yet, it does not offer an alternative modularization solution designed in synergy with multitier abstractions. Unfortunately, simply adopting traditional modularization mechanisms (e.g., a Haskell module for a Haskell-based multitier language) is not sufficient because such modularization mechanism needs to be aware of multitier code. Current approaches require whole-program analysis for slicing the applications (e.g., Ur/Web [Chlipala 2015]). Also, approaches that support separate compilation are still restricted to the client–server model, i.e., they do not support modular abstractions over the distributed architecture (e.g., Eliom [Radanne and Vouillon 2018]).

Summary Developing sophisticated distributed systems warrants language support for specifying the components of the distributed system and the communication among them, supporting proper modularization for large distributed systems. We draw inspiration from multitier programming, which helps in taming the complexity of developing distributed systems. Yet, existing approaches do not address the scope of problems of the development of generic distributed systems because multitier languages only support web applications rather than distributed systems in general. The insufficiencies described above hinder the development of distributed systems and demand a language design that considers the challenges in the domain of generic distributed applications from the ground up.

In this dissertation, we present our approach based on *placement types* to provide a coherent programming model for developing complex distributed systems with data flows spanning over multiple hosts.

1.2 ScalaLoci in Essence

We propose ScalaLoci, a programming language with a set of novel language features dedicated to ease the development of generic distributed applications. First, we propose *placement types* to associate locations to data and to computations. Our solution allows going beyond the web domain and enables static reasoning about

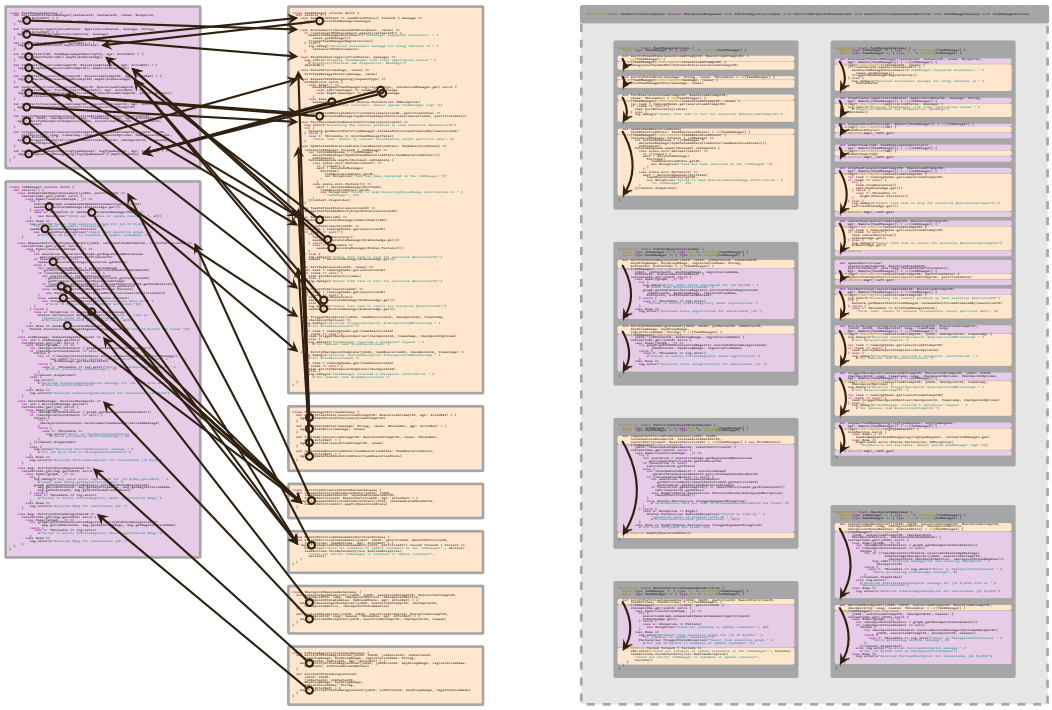
placement. ScalaLoci provides language-level features to declare distributed architectures for the system by specifying components and their relations. Second, for communication among hosts, we support *multitier reactives* – placed abstractions for reactive programming – to compose data flows spanning over multiple distributed components. Third, we design a *multitier module system* for ScalaLoci. Multitier modules encapsulate the interaction between distributed components of (sub)systems, allowing for (i) decoupling modularization from distribution and (ii) defining reusable patterns of interaction that model the functionality of a (distributed) subsystem and can be composed to build larger distributed systems. Our module system supports strong interfaces [Liskov 1987] to achieve encapsulation and information hiding, such that implementations can be easily exchanged.

1.3 ScalaLoci at Work

As an example for the application design enabled by ScalaLoci compared to a traditional design for a distributed system, Figure 1.1 on the following page provides an overview of the *task distribution system* in the Apache Flink stream processing framework [Carbone et al. 2015]. Apache Flink is used in production by many companies, including Amazon AWS, eBay, Ericsson, Comcast, Capital One or Huawei [Apache Software Foundation 2011a]. It consists of the coordinator of the Flink instance, the *JobManager* and one or more *TaskManagers*, which execute computational tasks.

Figure 1.1a on the next page shows the JobManager (dark violet boxes on the left), the TaskManager (light orange boxes on the right) and their communication (arrows). Every box is a class or an actor which is confined by network boundaries. Thus, cross-host data flow belonging to the same (distributed) functionality is scattered over multiple modules.

Figure 1.1b on the following page shows an implementation of the same system in ScalaLoci. We organize the code base into six multitier modules (the six deep gray boxes), where each module encapsulates a different feature of the task distribution system. The modules implemented in ScalaLoci are themselves distributed, i.e., they contain code for the JobManager (dark violet) and the TaskManager (light orange). Hence, modularization is not along network boundaries but to separate different (distributed) functionalities. The six modules are composed into the complete task distribution system (the box in the background). The data flow (arrows) in the system is not scattered over different modules and is much more *regular* due to the reorganization of the code that belongs to the same feature into the same distributed multitier module. The figures are discussed in more details in Sections 7.2.1 and 7.4.3 on page 126 and on page 144.



(a) Original Flink. **(b)** ScalaLoci Flink.

Figure 1.1 Communication and modularization for two actors in Flink.

1.3.1 Design Overview

Based on the ideas outlined above, we now sketch the main design aspects of ScalaLoci, which is designed as an extension to Scala. We present the design in full details in the rest of this dissertation.

The ScalaLoci multitier language supports generic distributed architectures. Developers can freely define the different components, called *peers*, of the distributed system. Peers are defined as abstract type members:

```
1 @peer type Registry
2 @peer type Node
```

Traits, classes and objects can define type members, which are either abstract (e.g., `type T`) or define concrete type aliases (e.g., `type T = Int`). Abstract types can define lower and upper type bounds (e.g., `type T >: LowerBound <: UpperBound`), which refine the type while keeping it abstract. We use annotations (i.e., `@peer`) to distinguish peer types from other type member definitions.

We further use peer types to express the architectural relation between the different peers by specifying *ties* between peers. Ties statically approximate the run time connections between peers. For example, a tie from a `Registry` peer to a `Node` peer

defines that, at run time, `Registry` instances can connect to `Node` instances. A *single* tie expresses the expectation that a single remote instance is always accessible.

Remote access is statically checked against the architectural scheme specified through ties. Hence, ties are also encoded at the type level such that that compiler can check that the code conforms to the specified architecture. Ties are defined by specifying a type refinement for peer types that declares a `Tie` type member:

```
@peer type Registry <: { type Tie <: Multiple[Node] }
@peer type Node <: { type Tie <: Single[Registry] with Multiple[Node] }
```

The type refinement `{ type Tie <: Multiple[Node] }` specified as upper bound for `Registry` states that `Registry` is a subtype of a type that structurally contains the definition of the `Tie` type member `type Tie <: Multiple[Node]`. The tie specification above defines (i) a multiple tie from the `Registry` to the `Node` peer and (ii) a single tie from the `Node` to the `Registry` peer as well as a multiple tie from the `Node` to the `Node` peer.

Having defined the components and their architectural relation using peer types, developers can place values on the different peers through *placement types*. The placement type `T on P` represents a value of type `T` on a peer `P`. The snippet places an integer on the registry:

```
val i: Int on Registry = 42
```

Accessing remote values requires the `asLocal` marker, creating a local representation of the remote value by transmitting it over the network:

```
val j: Future[Int] on Node = i.asLocal
```

Calling `i.asLocal` returns a future of type `Future[Int]`, accounting for network delay and potential communication failure. Futures represent values that will become available in the future or produce an error.

The approach of defining remote access via `asLocal` paves the way for specifying cross-host data flows in a declarative way. An `Event[T]` – representing a stream of discrete occurrences – results in a local representation of the event that fires whenever the remote event fires:

```
val sourceStream: Event[String] on Registry = Event[String]()
val remoteStream: Event[String] on Node = s.asLocal
```

Besides data flow specification, ScalaLoci also supports more traditional remote procedure calls. A remote procedure is invoked using `remote call`. If the result is of interest to the local instance, it can be made available locally using `asLocal`:

```
def square(i: Int): Int on Registry = i * i
val squareOfFour: Int on Node = (remote call square(4)).asLocal
```

ScalaLoci multitier code resides in *multitier modules*, i.e., in classes, traits or objects that carry the `@multitier` annotation. Multitier modules can be combined using mixin composition on traits or by referencing instances of multitier modules. Considering the following architecture specification for a module which provides a peer that monitors other peers (e.g., using a heartbeat mechanism):

```
1 @multitier trait Monitoring {
2   @peer type Monitor <: { type Tie <: Multiple[Monitored] }
3   @peer type Monitored <: { type Tie <: Single[Monitor] }
4 }
```

The following module reuses the monitoring functionality by defining an `object` extending the `Monitoring` trait to instantiate the `Monitoring` module (Line 2):

```
1 @multitier trait P2P {
2   @multitier object mon extends Monitoring
3
4   @peer type Registry <: mon.Monitor {
5     type Tie <: Multiple[mon.Monitored] with Multiple[Node] }
6   @peer type Node <: mon.Monitored {
7     type Tie <: Single[mon.Monitor] with Single[Registry] with Multiple[Node] }
8 }
```

The `P2P` module defines the `Registry` peer to be a special `Monitor` peer (Line 4) and the `Node` peer to be a special `Monitored` peer (Line 6) by declaring a subtype relation (e.g., `Registry <: mon.Monitor`) to map the architecture of the `Monitoring` module to the architecture of the `P2P` module, reusing the monitoring functionality. By defining that a `Registry` peer *is a* `Monitor` peer, all values placed on `Monitor` are also available on `Registry`. We use the path-dependent type `mon.Monitor` to refer to the `Monitor` peer of the multitier module instance `mon`. Types can be dependent on an path (of objects). Hence, we can distinguish between the peer types of different multitier module instances, i.e., the type members defined in different objects.

1.4 Contributions

Thanks to the combination of features described above, we believe that ScalaLoci provides a significant advance in tackling the complexity of distributed system development. In summary, this dissertation makes the following contributions:

- We introduce *placement types*, a novel language abstraction to associate locations to data and to computations, which allows distributed applications to be conceived as a whole, reasoning about the entire system and providing cross-host type-safety.

- We present ScalaLoci's design based on the core concept of placement types to support distributed applications, including in-language specification of distributed architectures, and specifying data flows over multiple hosts.
- We develop a core calculus for ScalaLoci with a type system to check that the interaction of remote values and local values is sound and that the application does not violate the architectural specification. We mechanize the proofs in Coq [Coq Development Team 2016].
- We present a multitier module system for ScalaLoci, which supports strong interfaces and exchangeable implementations. We show that *abstract peer types* enable a number of powerful abstractions to define and compose distributed systems, including multitier mixin composition and constrained modules.
- We present ScalaLoci's modular architecture where network and value propagation details are abstracted by *communicators* and *transmitters*, which support different communication mechanisms, e.g., TCP, WebSocket or WebRTC, and different reactive systems, e.g., REScala [Salvaneschi et al. 2014b] or Rx [Meijer 2010].
- We provide an implementation based on Scala that compiles multitier code to distributed components as specified through placement types, amounting to ~ 9 K lines of code for the language implementation, ~ 3 K lines of code for the language runtime and ~ 3 K lines of code to integrate different network protocols and reactive systems.
- We evaluate our approach with case studies – including distributed algorithms, distributed data structures, the Apache Flink stream processing framework, the Apache Gearpump real-time streaming engine and 22 variants of smaller case studies taken from different domains such as games, collaborative editing and instant messaging – showing that ScalaLoci applications exhibit better design and are safer and demonstrating the composition properties of multitier modules and how they can capture (distributed) functionalities in complex systems.
- We evaluate the run time performance of our language implementation. Microbenchmarks and system benchmarks on an Amazon EC2 distributed deployment show that ScalaLoci's advantages come at negligible performance cost.

In addition to contributing to the field of designing programming languages for distributed systems, ScalaLoci is also an experiment in implementing a distributed programming language as an embedded domain-specific language. To the best of our knowledge, our implementation is the largest case study of Scala macros.

1.5 Publications

The contributions of this dissertation have been published in peer-reviewed conferences and journals. Parts of the publications are used verbatim. We list the publications and which chapters of this dissertation they constitute:

Weisenburger, Pascal and Köhler, Mirko and Salvaneschi, Guido. 2018a. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2 (OOPSLA), Article 129. Boston, MA, USA. ACM, New York, NY, USA, 30 pages. ISSN: 2475-1421. DOI: 10.1145/3276499.
[Used in the abstract and Chapters 1 to 3, 5, 7 and 8]

Weisenburger, Pascal and Salvaneschi, Guido. 2019a. Multitier modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)*. Volume 134, Article 3. Leibniz International Proceedings in Informatics (LIPIcs). London, UK. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29 pages. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.3.
[Used in the abstract and Chapters 1 to 4, 6 and 8]

Weisenburger, Pascal and Salvaneschi, Guido. 2020. Implementing a language for distributed systems: Choices and experiences with type level and macro programming in Scala. *The Art, Science, and Engineering of Programming* 4 (3), Article 17. AOSA, Inc, 29 pages. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/17.
[Used in Chapters 1, 2, 6 and 8]

Weisenburger, Pascal and Wirth, Johannes and Salvaneschi, Guido. 2020. A survey of multitier programming. *ACM Computing Surveys* 53 (4), Article 81. ACM, New York, NY, USA, 35 pages. ISSN: 0360-0300. DOI: 10.1145/3397495.
[Used in Chapter 2]

I further (co)authored the following peer-reviewed publications and workshop papers, which are not part of this thesis:

Weisenburger, Pascal. 2016. Multitier reactive abstractions. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion '16. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 18–20. ISBN: 978-1-4503-4437-1. DOI: 10.1145/2984043.2984051.

Weisenburger, Pascal and Salvaneschi, Guido. 2016. Towards a comprehensive multitier reactive language. In *3rd International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '16. Amsterdam, Netherlands.

Weisenburger, Pascal and Luthra, Manisha and Koldehofe, Boris and Salvaneschi, Guido. 2017. Quality-aware runtime adaptation in complex event processing. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '17. Buenos Aires, Argentina. IEEE Press, Piscataway, NJ, USA, pages 140–151. ISBN: 978-1-5386-1550-8. DOI: 10.1109/SEAMS.2017.10.

Luthra, Manisha and Koldehofe, Boris and Weisenburger, Pascal and Salvaneschi, Guido and Arif, Raheel. 2018. TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*. DEBS '18. Hamilton, New Zealand. ACM, New York, NY, USA, pages 136–147. ISBN: 978-1-4503-5782-1. DOI: 10.1145/3210284.3210292.

Weisenburger, Pascal and Reinhard, Tobias and Salvaneschi, Guido. 2018b. Static latency tracking with placement types. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA/ECOOP Companion '18. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 34–36. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236486.

Weisenburger, Pascal and Salvaneschi, Guido. 2018. Multitier reactive programming with ScalaLoci. In *5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '18. Boston, MA, USA.

Mogk, Ragnar and Weisenburger, Pascal and Haas, Julian and Richter, David and Salvaneschi, Guido and Mezini, Mira. 2018b. From debugging towards live tuning of reactive applications. In *2018 LIVE Programming Workshop*. LIVE '18. Boston, MA, USA.

Weisenburger, Pascal. 2019. Demo: Developing distributed systems with ScalaLoci. In *Demo Track of the 3th International Conference on the Art, Science, and Engineering of Programming*. Programming Demos '19. Genoa, Italy.

Weisenburger, Pascal and Salvaneschi, Guido. 2019b. Tutorial: Developing distributed systems with multitier programming. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*. DEBS '19. Darmstadt, Germany. ACM, New York, NY, USA, pages 203–204. ISBN: 978-1-4503-6794-3. DOI: 10.1145/3328905.3332465.

Baumgärtner, Lars and Höchst, Jonas and Lampe, Patrick and Mogk, Ragnar and Sterz, Artur and Weisenburger, Pascal and Mezini, Mira and Freisleben, Bernd. 2019. Smart street lights and mobile citizen apps for resilient communication

in a digital city. In *Proceedings of the 2019 IEEE Global Humanitarian Technology Conference*. GHTC '19. Seattle, WA, USA. IEEE Press, Piscataway, NJ, USA. ISBN: 978-1-7281-1781-2. DOI: 10.1109/GHTC46095.2019.9033134.

Blöcher, Marcel and Eichholz, Matthias and Weisenburger, Pascal and Eugster, Patrick and Mezini, Mira and Salvaneschi, Guido. 2019. GRASS: Generic reactive application-specific scheduling. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '19. Athens, Greece. ACM, New York, NY, USA, pages 21–30. ISBN: 978-1-4503-6986-2. DOI: 10.1145/3358503.3361274.

1.6 Overview

Chapter 2 provides a survey of multitier languages that categorizes them under the aspects of their approach of defining placement, cross-host communication and supported distribution topologies. We also include ScalaLoci in the survey to compare its point in the design space to related approaches. Chapter 3 presents ScalaLoci's language abstractions and demonstrates them through examples. Chapter 4 describes the design of multitier modules and provides examples of how (sub)systems can be encapsulated and composed. Chapter 5 discusses the formalization of placement types and soundness proofs for placement and remote access. Chapter 6 outlines the ScalaLoci implementation regarding the type-level encoding of placement, the code splitting approach and the runtime system. Chapter 7 evaluates ScalaLoci's the design and performance. Chapter 8 concludes and outlines future research directions.

2 A Survey of Multitier Programming

> *And while the soulless minions of orthodoxy refuse to follow up on his important research, I could hear the clarion call of destiny ringing in my ears.*
>
> **— Elias Giger**

In this chapter, we provide an overview of multitier languages and of the fundamental design decisions that this paradigm entails. Multitier languages occupy different points in the design space, mixing techniques (e.g., compile time vs. run time splitting) and design choices (e.g., placement of compilation units vs. placement of single functions) that often depend on the application domain as well as on the software application stack. After presenting a selection of influential multitier languages, we systematically analyze existing multitier approaches along various axes, highlighting the most important achievements for each language. We also include the ScalaLoci multitier language presented in this thesis in the analysis to compare its point in the design space to related approaches. Finally, we provide an overview of research areas related to multitier approaches in general and to ScalaLoci's approach in particular.

2.1 Background

Developing distributed systems is widely recognized as a complex and error-prone task. A number of aspects complicate programming distributed software, including concurrent execution on different nodes, the need to adopt multiple languages or runtime environments (e.g., JavaScript for the client and Java for the server) and the need to properly handle complex communication patterns considering synchronicity/asynchronicity, consistency as well as low-level concerns such as data serialization and format conversion. Over the years, developers and practitioners have tackled these challenges with methods that operate at different levels. Various middlewares abstract over message propagation (e.g., Linda [Gelernter 1985]). Primitives for remote communication (e.g., CORBA [Group 1993], RMI [Wollrath

et al. 1996]) give programmers the illusion of distribution transparency. Decoupling in the software architecture improves concurrency and fault tolerance (e.g., the Actor model [Hewitt et al. 1973]). Finally, out-of-the-box specialized frameworks can manage fault recovery, scheduling and distribution automatically (e.g., MapReduce [Dean and Ghemawat 2008]).

A radically innovative solution has been put forward by the so-called *multitier programming* approach (sometimes referred to as *tierless programming*). Multitier programming consists of developing the components that pertain to different *tiers* in the system (e.g., client and server), mixing them in the same compilation unit. Code for different tiers is generated at run time or split by the compiler into components that belong to different tiers based on user annotations and static analysis, types or a combination of these.

A number of multitier *research* languages have been proposed over the last decade [Chlipala 2015; Cooper et al. 2006; Serrano et al. 2006], demonstrating the advantages of this paradigm, including improving software comprehension, enhancing software design, enabling formal reasoning and ameliorating maintenance. In parallel, a number of *industrial* solutions include concepts from multitier programming [Balat 2006; Bjornson et al. 2010; Strack 2012], showing that this approach has great potential in practice.

2.2 Multitier Programming

The different components of a distributed application are executed on different tiers, where each tier can run on a different machine in a network. For example, a 3-tier (or 3-layer) application is organized into three major parts – usually *presentation*, *application processing*, and *data management* – residing in different network locations [Buschmann et al. 1996]. One of the advantages of this approach is that, by organizing a system into tiers, the functionality that is encapsulated into one of the tiers can be modified independently, instead of redesigning the entire application as a whole.

As a result of this architectural choice, however, a crosscutting functionality that belongs to multiple tiers is separated among several compilation units. For example, in the Web setting, functionality is often scattered across client and server. Also, in many cases, each layer is implemented in a different programming language depending on the technology of the underlying layer, e.g., JavaScript for the browser-based interface, Java for the server-side application logic and SQL for the database.

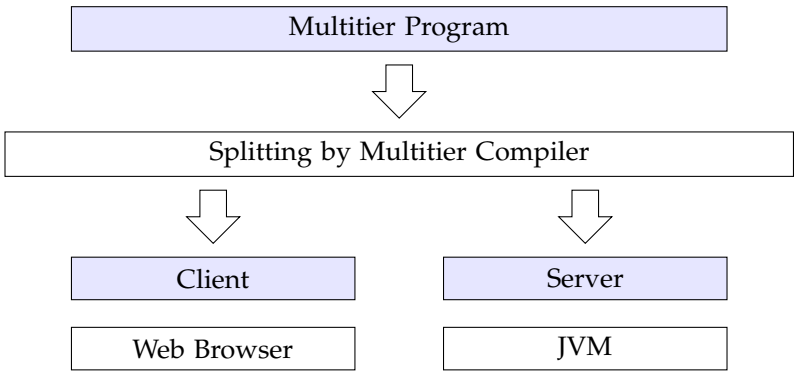


Figure 2.1 Multitier programming.

In a multitier programming language, a single language can be used to program different tiers, often adopting different compilation backends based on the target tier (e.g., JavaScript for the browser, Java for the server). As a result, a functionality that spans over multiple tiers can be developed within the same compilation unit. The compiler takes care of separating the code that belongs to different tiers and generating multiple deployable units starting from a single multitier program as well as of generating the network communication code that is required for such modules to interact during program execution (Figure 2.1).

2.3 Benefits

In this section, we provide an overview of the main advantages offered by the multitier language design. We report the main claims found in literature and refer to the sources where these are discussed.

Higher abstraction level An important advantage of multitier programming is that it enables abstracting over a number of low-level details relevant to programming distributed systems. As a result, software development is simplified and programmers can work at a higher level of abstraction [Weisenburger et al. 2018a]. Specifically, developers do not face the issue of dealing with error-prone aspects like network communication, serialization, and data format conversions between different tiers [Radanne et al. 2016]. Further, with multitier programming, there is no need to design the inter-tier APIs, for example specifying the REST API which a server exposes to clients. The technologies used for inter-tier communication are usually transparent to the developer [Serrano et al. 2006] and a detail of the compilation approach.

Improved software design In many distributed applications, the boundaries between hosts and the boundaries between functionalities do not necessarily coincide, i.e., a single functionality can span multiple locations and a single location can host multiple functionalities. For example, retrieving a list of recent emails requires a search on the server, filtering the result on the client and displaying the result. These operations conceptually pertain to the same functionality. Programming each location separately may result in two design issues: First, it can compromise modularity because functionality (e.g., email retrieval) is scattered across the codebases of different hosts. Second, it is error-prone because of code repetition. For example, encryption requires encrypting and decrypting data on both ends of the communication channel, and the associated functions need to be available on both the client and the server. In contrast, multitier programming allows for developing a functionality once and then place it where required [Delaval et al. 2008].

Formal reasoning Formal reasoning can benefit from multitier design because multitier languages model distributed applications as a whole as well as reify a number of aspects of distributed software that are usually left implicit, like placement, components of the distributed system, and the boundaries among tiers. Hence, it becomes easier to formally reason about software properties considering the whole system at once instead of each component in isolation. For example, researchers have developed methods to reason about concurrency [Neubauer and Thiemann 2005] and security [Baltopoulos and Gordon 2009] considering information flow in the whole system. Also, performance can be improved by eliminating dynamic references of global pointers [Chandra et al. 2008]. Finally, researchers considered domain-specific properties, such as reachability in software defined networks via verification [Nelson et al. 2014].

Code maintenance Multitier programming simplifies the process of modifying an existing software system. Two cases are particularly interesting: First, migrating functionality among different tiers does not require a complete rewrite in a different language [Groenewegen et al. 2008]. For example, validating user input should already happen on the client-side to improve usability and must happen on the server to enforce input validation before further processing. Both validation functions share the same code. Second, it is easier to migrate an application among different platforms [Haxe Foundation 2005]. For example, in principle, the client-side logic of a client–server desktop application can be migrated to the Web just by changing the compilation target of the client side to JavaScript.

Program comprehension Program comprehension refers to the complexity (time, required expertise) that a programmer faces to develop a correct mental model of the behavior of a program [Soloway and Ehrlich 1984]. A crucial advantage of multitier programming is that it simplifies reasoning about data flow over multiple hosts because data flows that belong to a certain functionality are not interrupted by the modularization across the tier axis and by the details of communication code – simplifying development as well as debugging [Manolescu et al. 2008]. We are, however, not aware of empirical studies or controlled experiments that measure the advantage of multitier programming in terms of program comprehension.

2.4 Overview

This section provides an overview over multitier languages, i.e., languages that support implementing different tiers of a distributed system within a single compilation unit. We focus on *homogeneous* multitier programming, where tiers follow the same model of computation and have similar processing capabilities. Databases are an example for a tier with a computational model that is typically different from the one of the tier that accesses the database, such as a web server. For multitier languages that support *heterogeneous* tiers, such as databases, we only briefly describe the language features that are supported. Table 2.1 on the following page lists the multitier approaches we discuss systematically and related approaches on which we touch to point out their connection to multitier programming.

Multitier languages In the following survey, we first show the implementation of a small application (Section 2.5 on page 20) in a representative selection of multitier languages. These include two languages that pioneered multitier programming for the web (Hop/Hop.js and Links), two recent approaches focusing on web development (Ur/Web and Eliom), our approach that also supports more general distributed systems than web applications (ScalaLoci) and Google's GWT, an industrial solution for cross compilation to different tiers, that, however, provides no specific multitier abstractions. We then conduct a systematic feature comparison (Section 2.6 on page 29) among homogeneous multitier languages (Table 2.1 on the next page).

We also include programming frameworks that target distributed applications where several tiers are developed together, using the same language (Table 2.1 on page 19, upper segment). For example, such frameworks reuse existing (non-multitier) languages and communication libraries, compiling to JavaScript for the client-side (GWT), using JavaScript for both the client and the server (Meteor) or

Table 2.1 Overview of multitier languages.

Language	Short Description
Hop/Hop.js [Serrano et al. 2006]/ [Serrano and Prunet 2016]	Dynamically typed language for developing web applications with a client–server communication scheme and asynchronous callbacks.
Links [Cooper et al. 2006]	Statically typed language that covers the client tier, the server tier and the access to the database tier. It uses remote calls and message passing for client–server communication.
Ur/Web [Chlipala 2015]	ML-like language with support for type-safe metaprogramming that provides communication from client to server through remote procedure calls and from the server to the client through message-passing channels.
Eliom/Ocsigen [Radanne et al. 2016]/ [Balat 2006]	OCaml dialect that extends the ML module system to support multitier modules featuring separate compilation; used in the Ocsigen project.
ScalaLoci [Weisenburger et al. 2018a]	Supports generic distributed systems, not only web applications, thanks to placement types; features remote procedures and reactive programming abstractions for remote communication.
StiP.js [Philips et al. 2018]	Allows developers to annotate the code that belongs to the client or to the server; slicing detects the dependencies between the annotated fragment and the rest of the code.
Gavial [Reynders et al. 2020]	Domain-specific language embedded into Scala that provides reactive programming abstractions for cross-tier communication.
Opa [Rajchenbach-Teller and Sinot 2010]	Statically typed language that supports remote communication via remote procedure calls and message-passing channels.
AmbientTalk/R [Dedecker et al. 2006]/ [Carreton et al. 2010]	Targets mobile applications with loosely coupled devices and provides reactive programming abstractions on top of a publish–subscribe middleware.
ML5 [Murphy et al. 2007]	Represents different tiers by different possible worlds, as known from modal logic.
WebSharper [Bjornson et al. 2010]	Allows developers to specify client-side members and members that are callable remotely.
Haste [Ekblad and Claessen 2014]	Uses monadic computations wrapping client and server code into different monads and provides explicit remote calls.
Fun [Westin 2010]	Enables automatic synchronization of data across web clients without manually implementing the communication with the server.
Koka [Leijen 2014]	Supports splitting code among tiers using a type and effect system by associating different effects to different tiers.
Multi-Tier Calculus [Neubauer and Thiemann 2005]	Provides a formal model to reason about the splitting of multitier code into a client and a server part and the communication between both parts through message channels.
Swift [Chong et al. 2007a]	Splits an application into client and server programs based on the flow of private data, making sure that private data does not flow to untrusted clients.
Volta [Manolescu et al. 2008]	Uses attributes to annotate classes with the tier they belong to, automatically converting cross-tier method calls to remote invocations.

Table 2.1 Overview of multitier languages (continued).

Language	Short Description
GWT [Kereki 2010]	Compiles Java to JavaScript for the client and provides remote procedures for client–server communication; developed at Google.
Meteor [Strack 2012]	A programming framework to use JavaScript for both the client and the server code; provides remote procedures, publish–subscribe abstractions and shared state.
J-Orchestra [Tilevich and Smaragdakis 2002]	Uses configuration files to assign Java classes to tiers, rewriting the Java bytecode to turn method invocations into remote calls.
Hiphop [Berry et al. 2011]	Extends Hop with synchronous data flows, focusing on guarantees on time and memory bounds.
Distributed Orc [Thywissen et al. 2016]	The runtime optimizes the placements of values; it provides location transparency by giving local and remote operations the same semantics, which allows for handling asynchrony and failures uniformly.
Jif/split [Zdancewic et al. 2002]	Splits a program into tiers based on the flow of private data, making sure that private data do not flow to another tier.
Fission [Guha et al. 2017]	Dynamically splits a program execution into client-side and server-side execution based on the flow of private data, making sure that private data does not flow to untrusted clients.
SIF [Chong et al. 2007b]	Checks the flow of private data in a web application, making sure that private data does not flow to untrusted clients.
WebDSL [Groenewegen et al. 2008]	Domain-specific language for specifying the data model of web applications and the web pages to view and edit data model objects.
Acute [Sewell et al. 2005]	Supports type-safe marshalling for remote interaction, versioning of program code and dynamic code reloading, leaving the network communication mechanism to libraries.
Mobl [Hemel and Visser 2011]	Supports different concerns of developing the client-side of web applications, such as the data model, the application logic and the user interface.
High-Level Abstractions for Web Programming [Richard-Foy et al. 2013]	Provides a Scala EDSL that captures common tasks performed in web applications, e.g., defining DOM fragments.

use an external configuration file for specifying the splitting (J-Orchestra). In these languages, the presence of different tiers is clearly visible to the programmer either in the form of configuration files or source annotations.

Related approaches We also elaborate on closely related approaches (Table 2.1, lower segment) that do not completely fit the programming model of the aforementioned multitier languages and the taxonomy of our feature comparison. Hence, we do not classify them systematically but highlight their connection to multitier programming where they relate to the discussed multitier aspects. Such approaches (a) do not express tiers as part of their language abstractions because the code

is assigned to tiers transparently (Distributed Orc, Jif/split and Fission). In this group, we also include Hiphop, where the language extends a multitier language but the extension itself does not add any multitier abstractions, and SIF, which uses GWT for JavaScript compilation as well as a client runtime library, and WebDSL, where the language only represents the state of the data model. Other approaches do not completely fit the multitier programming model that we consider because they (b) do not include cross-tier communication, intentionally leaving remote communication support to libraries, such as Acute and several languages for web applications, such as Mobl and High-Level Abstractions for Web Programming.

Multitier development shares the goal of abstracting over different tiers with cross-compilation, which also abstracts over the heterogeneity of different target platforms. Cross-compilers include, e.g., Haxe or the Kotlin language, the JSweet Java to JavaScript compiler, the Bridge.NET and the SharpKit C# to JavaScript compilers and the Scala.js Scala to JavaScript compiler. Yet, these solutions do not offer specific language-level support for distribution and remote communication. We discuss the difference between cross-compilers and multitier languages, but do not consider cross-compilers in detail.

2.5 A Glimpse of Multitier Languages

In this section, we present languages that have pioneered multitier programming and/or have been very influential in recent years. To provide an intuition of how multitier programming looks like using those languages, we present the same example implemented in each of these languages. As an example, we show an *Echo* client–server application: The client sends a message to the server and the server returns the same message to the client, where it is appended to a list of received messages. The application is simple and self-contained and – despite all the limitations of short and synthetic examples – it gives us the chance to demonstrate different multitier languages side by side.

2.5.1 Hop

Hop [Serrano et al. 2006] is a dynamically typed Scheme-based language. It follows the traditional approach of modeling communication between client and server using asynchronous callbacks for received messages and return values. JavaScript code is generated at run time and passed to the client. A recent line of work has ported the results of Hop to a JavaScript-based framework, Hop.js [Serrano and Prunet 2016], which allows using JavaScript to program both the client and the server side.

Listing 2.1 Echo application in Hop.js.

```
1  service echo() {
2    var input = <input type="text" />
3    return <html>
4      <body onload=~{
5        var ws = new WebSocket("ws://localhost:" + ${hop.port} + "/hop/ws")
6        ws.onmessage = function(event) {
7          document.getElemenetById("list").appendChild(<li>${event.data}</li>)
8        }
9      }>
10       <div>
11         ${input}
12         <button onclick=~{ ws.send(${input}.value) }>Echo!</button>
13       </div>
14       <ul id="list" />
15     </body>
16   </html>
17 }
18
19 var wss = new WebSocketServer("ws")
20 wss.onconnection = function(event) {
21    var ws = event.value
22    ws.onmessage = function(event) { ws.send(event.value) }
23 }
```

Listing 2.1 shows the Echo application implemented in Hop.js. HTML can be embedded directly in Hop code. HTML generated on the server (Lines 2–16) is passed to the client. HTML generated on the client can be added to the page using the standard DOM API (Line 7). Hop supports bidirectional communication between a running server and a running client instance through its standard library. In the Echo application, the client connects to the WebSocket server through the standard HTML5 API (Line 5) and sends the current input value (Line 12). The server opens a WebSocket server (Line 19) that returns the value back to the client (Line 22). The language allows the definition of *services*, which are executed on the server and produce a value that is returned to the client which invoked the service. For example, the `echo` service (Line 1) produces the HTML page served to the web client of the Echo application. Thus, the code in a service block is executed on the server. By using the `~{...}` notation, the code for the `onload` (Line 4) and `onclick` (Line 12) handlers is not immediately executed but the server generates the code for later execution on the client. On the other hand, the `${...}` notation escapes one level of program generation. The expressions `hop.port` (Line 5), `event.data` (Line 7) and `input` (Lines 11 and 12) are evaluated by the outer server program and the values to which they evaluate are injected into the generated client program.

Listing 2.2 Echo application in Links.

```
1  fun echo(item) server {
2    item
3  }
4
5  fun main() server {
6    page
7      <html>
8        <body>
9          <form l:onsubmit="{
10           appendChildren(<li>{stringToXml(echo(item))}</li>,
11             getNodeById("list"))
12         }">
13           <input l:name="item" />
14           <button type="submit">Echo!</button>
15         </form>
16         <ul id="list" />
17       </body>
18     </html>
19 }
20
21 main()
```

Hop supports full stage programming, i.e., ~{...} expressions can be arbitrarily nested such that not only server-side programs can generate client-side programs but also client-side programs are able to generate other client-side programs.

2.5.2 Links

Links [Cooper et al. 2006] is a statically typed language that translates to SQL for the database tier and to JavaScript for the web browser. The latter is a technique, which was pioneered by the typed query system Kleisli [Wong 2000] and adopted by Microsoft LINQ [Torgersen 2007]. It allows embedding statically typed database queries in Links. Recent work extended Links with algebraic effects [Hillerström et al. 2017], provenance tracking [Fehrenbach and Cheney 2019] and session types [Lindley and Morris 2017] with support for exception handling [Fowler et al. 2019].

Listing 2.2 shows the Echo application implemented in Links. Links uses annotations on functions to specify whether they run on the client or on the server (Lines 1 and 5). Upon request from the client, the server executes the main function (Line 21), which constructs the code that is sent to the client. Links allows embedding XML code (Lines 7–18). The `l:name` attribute (Line 13) declares an identifier to which the value of the input field is bound and which can be used elsewhere (Line 10). The code to be executed for the `l:onsubmit` handler (Line 9) is not immediately executed but compiled to JavaScript for client-side execution. Curly braces indicate

Links code embedded into XML. The `l:onsubmit` handler sends the current input value `item` to the server by calling `echo`. The item is returned by the server and appended to the list of received items using standard DOM APIs. The call to the server (Line 9) does not block the client. Instead, the continuation on the client is invoked when the result of the call is available. Client–server interaction is based on *resumption passing style*: Using continuation passing style transformation and defunctionalization, remote calls are implemented by passing the name of a function for the continuation and the data needed to continue the computation. Rather than of constructing HTML forms manually, like in the example, Links further supports *formlets* [Cooper et al. 2008], an abstraction for composing HTML forms.

To access the database tier, Links features database expressions to represent database connections. For example, to store the list of received items in a server-side database, the `table "items" with (item: String) from database "list"` expression refers to the *items* table in the *list* database that contains records with a single *item* string field. Links supports language constructs for querying and updating databases – such as iterating over records using `for`, filtering using `where` clauses, sorting using `orderby` or applying functions on lists, such as `take` and `drop`, to data sets – which are compiled into equivalent SQL statements.

2.5.3 Ur/Web

Ur/Web [Chlipala 2015] is a language in the style of ML, featuring an expressive type system to support type-safe metaprogramming. The type system ensures correctness of a broad range of properties including (i) validity of generated HTML code, (ii) the types of values of HTML form fields matching the types expected by their handlers or the types of columns of a database table, (iii) validity of SQL queries, (iv) lack of dead intra-application links and (v) prevention of code injection attacks. Further, Ur/Web prevents cross site request forgery attacks through the use of cryptographic signatures. Remote procedure calls are executed atomically, with Ur/Web guaranteeing the absence of observable interleaving operations.

Listing 2.3 on the following page shows the Echo application implemented in Ur/Web. Ur/Web allows embedding XML code using `<xml>...</xml>` (Lines 6 and 7). The `{...}` notation embeds Ur/Web code into XML. `{[...]}` evaluates an expression and embeds its value as a literal. Ur/Web supports functional reactive programming for client-side user interfaces. The example defines an `item` source (Line 9), whose value is automatically updated to the value of the input field (Line 13) when it is changed through user input, i.e., it is *reactive*. The `list` source (Line 10) holds the list of received items from the echo server. Sources, time-changing input values, and signals, time-changing derived values, are Ur/Web's

Listing 2.3 Echo application in Ur/Web.

```
1  fun echo (item : string) = return item
2
3  fun main () =
4    let fun mkhtml list =
5      case list of
6        []        => <xml/>
7      | r :: list => <xml><li>{[r]}</li>{mkhtml list}</xml>
8    in
9      item <- source "";
10     list <- source [];
11     return <xml><body>
12       <div>
13         <ctextbox source={item} />
14         <button value="Echo!" onclick={ fn _ =>
15           list' <- get list;
16           item' <- get item;
17           item' <- rpc (echo item');
18           set list (item' :: list')
19         }/>
20       </div>
21       <ul>
22         <dyn signal={
23           list' <- signal list;
24           return (mkhtml list')
25         }/>
26       </ul>
27     </body></xml>
28   end
```

reactive abstractions, i.e., signals recompute their values automatically when the signals or sources from which they are derived change their value, facilitating automatic change propagation. Upon clicking the button, the current value of `list` (Line 15) and `item` is accessed (Line 16), then a remote procedure call to the server's `echo` function is invoked (Line 17) and `list` is updated with the item returned from the server (Line 18). To automatically reflect changes in the user interface, a signal is bound to the `signal` attribute of the HTML pseudo element `<dyn>` (Line 22). The signal uses the `mkhtml` function (Line 24, defined in Line 4), which creates HTML list elements. In addition to remote procedure calls – which initiate the communication from client to server – Ur/Web supports typed message-passing channels, which the server can use to push messages to the client.

Ur/Web integrates a domain-specific embedding of SQL for accessing the database tier with clauses such as `SELECT`, `FROM` or `ORDERBY`. For example, the declaration `table items : { item : string }` specifies a set of database records storing the list of received items. Such `table` declarations can be private to a module using an ML-style module system for encapsulating database tables.

Listing 2.4 Echo application in Eliom.

```
1  module Echo_app = Eliom_registration.App (
2   struct let application_name = "echo" let global_data_path = None end)
3
4  let%server main_service = create
5   ~path:(Path []) ~meth:(Get Eliom_parameter.unit) ()
6
7  let%server make_input up =
8   let inp = Html.D.Raw.input () in
9   let btn = Html.D.button
10   ~a:[Html.D.a_class ["button"]] [Html.D.pcdata "Echo!"] in
11  ignore [%client
12   (Lwt.async (fun () ->
13    Lwt_js_events.clicks (Html.To_dom.of_element ~%btn) (fun _ _ ->
14     ~%up (Js.to_string (Html.To_dom.of_input ~%inp)##.value);
15     Lwt.return_unit)) : unit) ];
16  Html.D.div [inp; btn]
17
18 let%server () = Echo_app.register
19  ~service:main_service
20  (fun () () ->
21   let item_up = Up.create
22    (Eliom_parameter.ocaml "item" [%derive.json :string]) in
23   let item_down = Down.of_react (Up.to_react item_up) in
24   let list, handle = ReactiveData.RList.create [] in
25   let list = ReactiveData.RList.map
26    [%shared fun i -> Html.D.li [Html.D.pcdata i] ] list in
27   let input = make_input item_up in
28   ignore [%client
29    (Eliom_client.onload
30     (fun _ -> ignore (React.E.map
31      (fun i -> ReactiveData.RList.cons i ~%handle) ~%item_down)) : unit) ];
32    Lwt.return
33     (Eliom_tools.D.html ~title:"echo" (Html.D.body [input; Html.R.ul list])))
```

2.5.4 Eliom

Eliom [Radanne et al. 2016] is an OCaml dialect designed in the context of the Ocsigen project [Balat 2006] for developing client–server web applications. Ocsigen further provides mechanisms to support a number of practical features necessary in modern applications, including session management and bidirectional client–server communication through its standard library.

Listing 2.4 shows the Echo application in Eliom. Eliom extends `let`-bindings with *section annotations* `%client`, `%server` and `%shared` – the latter indicates code that runs on both the client and the server. The application starts with a call to `Echo_app.register` (Line 18). Eliom supports cross-tier reactive values: The application generates a server-side event (Lines 21 and 22) and a corresponding client-side event (Line 23), which automatically propagates changes from the server to the client. A reactive list (Line 24) holds the items received from the server.

Mapping the list produces a list of corresponding HTML elements (Line 25), which can directly be inserted into the generated HTML code (Line 33). Eliom supports a DSL for HTML, providing functions of the same name as the HTML element they generate. Server-side code can contain nested *fragments* to be run on the client (`[%client ...]`, Line 28) or to be run on both the client and the server (`[%shared ...]`, Line 26).

Eliom uses *injections* (prefixed by `~%`) to access values on the client side that were computed on the server. The client-side representation of the event `item_down` is injected into a client fragment to extend the reactive list with every item returned from the server (Line 31). The `make_input` function (Line 7) generates the main user interface, which processes the stream of button clicks (Line 13) and fires the `up` event for every item (Line 14). To fire the server-side `up` event from the client-side, we inject the event via `~%up` into the client fragment.

Client fragments are not run immediately when server code is evaluated. Instead, they are registered for later execution when the web page is delivered to the client. Hence, in the Eliom execution model, a single communication – from the server to the client – takes place.

2.5.5 Google Web Toolkit (GWT)

GWT [Kereki 2010] is an open source project developed at Google. Its design has been driven by a pragmatic approach, mapping traditional Java programs to web applications. A GWT program is a Java Swing application except that the source code is compiled to JavaScript for the client side and to Java bytecode for the server side. Compared to fully-fledged multitier programming, distributed code in GWT is not developed in a single compilation unit nor necessarily in the same language. Client and server code reside in different Java packages. Besides Java, in practice, GUIs often refer to static components in external HTML or XML files. GWT provides RPC library support for cross-tier communication.

Listing 2.5 on the facing page shows the Echo application implemented in GWT. For the sake of brevity, we leave out the external HTML file. The application adds an input field (Line 10.d) and a button (Line 11.d) to container elements defined in the HTML file and registers a handler for click events on the button (Line 13.d). When the button is clicked, the `echo` method of the `echoService` is invoked with the current item and a callback – to be executed when the remote call returns. When an item is returned by the remote call, it is added to the list of received items (Line 18.d). GWT requires developers to specify both the interface implemented by the service (Line 3.a) and the service interface for invoking methods remotely

Listing 2.5 Echo application in GWT.

```
1.a  package echo.client;
2.a
3.a  public interface EchoService extends RemoteService {
4.a    String echo(String item) throws IllegalArgumentException;
5.a  }
```

```
1.b  package echo.client;
2.b
3.b  public interface EchoServiceAsync {
4.b    void echo(String item, AsyncCallback<String> callback)
5.b      throws IllegalArgumentException;
6.b  }
```

```
1.c  package echo.server;
2.c
3.c  public class EchoServiceImpl extends RemoteServiceServlet
4.c      implements EchoService {
5.c    public String echo(String item) throws IllegalArgumentException {
6.c      return item;
7.c    }
8.c  }
```

```
1.d  package echo.client;
2.d
3.d  public class Echo implements EntryPoint {
4.d    private final EchoServiceAsync echoService = GWT.create(EchoService.class);
5.d
6.d    public void onModuleLoad() {
7.d      final TextBox itemField = new TextBox();
8.d      final Button submitButton = new Button("Echo!");
9.d
10.d     RootPanel.get("itemFieldContainer").add(itemField);
11.d     RootPanel.get("submitButtonContainer").add(submitButton);
12.d
13.d     submitButton.addClickHandler(new ClickHandler {
14.d       public void onClick(ClickEvent event) {
15.d         echoService.echo(itemField.getText(), new AsyncCallback<String>() {
16.d           public void onFailure(Throwable caught) { }
17.d           public void onSuccess(String result) {
18.d             RootPanel.get("itemContainer").add(new Label(result));
19.d           }
20.d         });
21.d       }
22.d     });
23.d   }
24.d 1 }
```

Listing 2.6 Echo application in ScalaLoci.

```
1  @multitier object Application {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4
5    val message = on[Client] { Event[String]() }
6    val echoMessage = on[Server] { message.asLocal }
7
8    def main() = on[Client] {
9      val items = echoMessage.asLocal.list
10     val list = Signal { ol(items() map { message => li(message) }) }
11     val inp = input.render
12     dom.document.body = body(
13       div(
14         inp,
15         button(onclick := { () => message.fire(inp.value) })("Echo!")),
16       list.asModifier).render
17   }
18 }
```

using a callback (Line 3.b). The implementation of the echo service (Line 3.c) simply returns the item sent from the client.

2.5.6 ScalaLoci

The ScalaLoci language presented in this thesis targets *generic* distributed systems rather than the Web only, i.e., it is not restricted to a client–server architecture. To this end, ScalaLoci supports peer types to encode the different locations at the type level. Placement types are used to assign locations to data and computations. ScalaLoci supports multitier reactives – language abstractions for reactive programming that are placed on specific locations – for composing data flows which span across different peers.

Listing 2.6 shows the Echo application implemented in ScalaLoci. The application first defines an input field (Line 11) using the ScalaTags library [Li 2012]. The value of this input field is used in the click event handler of a button (Line 15) to fire the `message` event with the current value of the input field. The value is then propagated to the server (Line 6) and back to the client (Line 9). On the client, the values of the event are accumulated using the `list` function and mapped to an HTML list (Line 10). This list is then used in the HTML code (Line 16) to display the previous inputs.

2.6 Analysis

In this section we systematically analyze existing multitier solutions along various axes. We consider the following dimensions:

Degrees of multitier programming refers to the amount of multitier abstractions supported by the language. At one extreme of the spectrum, we find languages with dedicated multitier abstractions for data sharing among tiers and for communication. At the other end of the spectrum lie languages where part of the codebase can simply be cross-compiled to a different target platform (e.g., Java to JavaScript) to enhance the interoperability between tiers but do not provide specific multitier abstractions.

Placement strategy describes how data and computations in the program are assigned to the hosts in the distributed system, e.g., based on programmers' decisions or based on automatic optimization.

Placement specification and granularity in multitier languages refers to the means offered for programmers to specify placement (e.g., code annotations, configuration files) and their granularity level (e.g., per function, per class).

Communication abstractions for communication among tiers are a crucial aspect in multitier programming since multitier programming brings the code that belongs to different tiers to the same compilation unit. Multitier approaches provide dedicated abstractions to simplify implementing remote communication which differ considerably among languages.

Formalization of multitier languages considers the approach used to formally define the semantics of the language and formally prove properties about programs.

Distribution topologies describe the variety of distributed system architectures (e.g., client–server, peer-to-peer) that a language supports.

2.6.1 Degrees of Multitier Programming

Several programming frameworks for distributed systems have been influenced, to various degrees, by ideas from multitier programming. In this section, we compare languages where multitier programming is supported by dedicated abstractions, either by explicitly referring to placement in the language or by using scoping in the same compilation unit to define remote communication, and approaches that share similar goals to multitier programming using compilation techniques that support

Table 2.2 Degrees of multitier programming.

Compilation Approach	**Distribution Approach**		
	Distribution		*No Distribution Abstractions*
	Multitier	*Transparent*	
Cross Compilation	Hop Links Opa Ur/Web Eliom/Ocsigen Gavial ML5 ScalaLoci WebSharper Haste Swift Volta GWT		Haxe Kotlin JSweet Bridge.NET SharpKit Scala.js WebDSL Mobl High-Level Abstractions for Web Programming
Uniform Compilation	Hop.js StiP.js AmbientTalk/R Fun Koka Multi-Tier Calculus J-Orchestra Meteor	Distributed Orc Jif/split Fission	Hiphop SIF Acute (traditional languages)

different targets (and tiers), but do not expose distribution as a language feature to the developer. Table 2.2 provides an overview of existing solutions concerning the degree of supported multitier programming. Specifically, it considers support for cross compilation and the supported language features for distribution.

Multitier distribution provides a programming model that defines different tiers and offers abstractions for developers to control the distribution.

Transparent distribution does not support code assignment to tiers as a reified language construct. Splitting into tiers is computed transparently by the compiler or the runtime and is not part of the programming model.

No distribution abstractions do not provide language features specific to the distribution of programs.

When running distributed applications on different machines, approaches related to multitier programming either assume the same execution environment, where all tiers can be supported by a **uniform compilation** scheme, or employ a **cross compilation** approach to support different target platforms. Cross-compilers can be used to support the development of distributed systems (e.g., by compiling client-side code to JavaScript) but still require manual distribution of code and do not offer abstractions for remote communication among components as multitier

languages do. Traditional languages, falling into the bottom right corner of Table 2.2 on the preceding page, neither support distribution nor cross compilation. Hiphop [Berry et al. 2011] does not provide its own support for distribution but relies on Hop's [Serrano et al. 2006] multitier primitives. SIF [Chong et al. 2007b] uses information flow control to ensure that private data does not flow to untrusted clients. It is implemented on top of Java Servlets, which respond to requests sent by web clients. Acute [Sewell et al. 2005] is an OCaml extension that, although it does not support distribution or cross compilation, provides type-safe marshalling for accessing resources remotely based on transmitting type information at run time for developing distributed systems.

We provide examples for the multitier category, which is extensively discussed in the rest of the survey, and systematically analyze the second and third approach using transparent splitting by the compiler or manual splitting and cross compilation, respectively.

Dedicated multitier programming abstractions Multitier languages provide abstractions that reify the placement of data and computations and allow programmers to directly refer to these concepts in their programs. In Hop.js [Serrano and Prunet 2016], inside the same expression, it is possible to switch between server and client code with `~{...}` and `${...}`, which can be arbitrarily nested. Similarly, the Ur/Web [Chlipala 2015] language provides the `{...}` escape operator. In ScalaLoci [Weisenburger et al. 2018a], placement is part of the type system (placement types) and the type checker can reason about resource location in the application. Eliom's [Radanne et al. 2016] placement annotations `%client`, `%server` and `%shared` allow developers to allocate resources in the program at the granularity of variable declarations. Similarly, Links [Cooper et al. 2006] provides a `client` and a `server` annotation to indicate functions that should be executed on the client or the server, respectively.

The multitier languages above hide the mismatch between the different platforms underlying each tier, abstracting over data representation, serialization and network protocols, enabling the combination of code that belongs to different tiers within the same compilation unit. In addition, multitier concepts are reified in the language in the sense that language abstractions enable developers to refer to tiers *explicitly*.

Transparent splitting Transparent distribution approaches enable using a single language for different tiers and support compilation to tier-specific code, but do not provide specific abstractions for multitier programming. Splitting a program into different tiers based on security concerns (Jif/split [Zdancewic et al. 2002], Fission [Guha et al. 2017]) adopts information flow control techniques to ensure

that private data does not leak to untrusted tiers. Distributed Orc [Thywissen et al. 2016] automatically optimizes the distribution of values at run time to minimize communication cost.

Cross-compilers Approaches that add compilation to a different platform for existing general-purpose languages have been proposed by different vendors and organizations, targeting various languages and programming platforms, e.g., the JSweet Java to JavaScript compiler, the Bridge.NET and the SharpKit C# to JavaScript compilers and the Scala.js Scala to JavaScript compiler. Haxe [Haxe Foundation 2005] is a cross-platform toolkit based on the statically typed object-oriented Haxe language that compiles to JavaScript, PHP, C++, Java, C#, Python and Lua. The statically typed language Kotlin [JetBrains 2009] for multi-platform applications targets the JVM, Android, JavaScript and native code. Such approaches do not support *automatic* separation into tiers – the developer has to keep the code for different tiers separate, e.g., in different folders. Remote communication APIs are provided by libraries depending on the target platform (e.g., TCP sockets or HTTP). Such solutions are the most pragmatic: They do not break compatibility with tooling – if already available – and provide a programming model that is quite close to traditional programming. Developers do not significantly change the way they reason about coding distributed applications and do not need to learn completely new abstractions.

Domain-specific languages take over tasks specific to (certain types of) distributed applications, such as marshalling data for remote transmission or constructing a client-side user interface based on a given data model. Richard-Foy et al. [2013] propose a Scala EDSL that captures common tasks performed in web applications, e.g., defining DOM fragments. Their approach allows specializing code generation depending on the target platform, e.g., using the Scala XML library when compiling to Java bytecode or using the browser's DOM API when compiling to JavaScript. Mobl [Hemel and Visser 2011] is a DSL for building mobile web applications in a declarative way providing language features for specifying the data model, the application logic and the user interface. Mobl compiles to a combination of different target languages, i.e., HTML, CSS and JavaScript. It, however, targets the client side only.

2.6.2 Placement Strategy

The placement strategy is the approach adopted by multitier languages to assign data and computations in the program to the hosts comprising the distributed system. Table 2.3 on the facing page classifies multitier languages into approaches

Table 2.3 Placement strategy.

Language	Placement Strategy	
	Automatic	Explicit
Hop/Hop.js	·	staged
Links	·	partitioned
Opa	partitioned	·
StiP.js	partitioned	·
Ur/Web	·	staged
Eliom/Ocsigen	·	staged
Gavial	·	partitioned
AmbientTalk/R	·	partitioned
ML5	·	partitioned
ScalaLoci	·	partitioned
WebSharper	·	partitioned
Haste	·	partitioned
Fun	·	partitioned
Koka	·	partitioned
Multi-Tier Calculus	partitioned	·
Swift	partitioned	·
Volta	·	partitioned
J-Orchestra	·	partitioned
Meteor	·	partitioned
GWT	·	partitioned

where placement is done **automatically** and approaches where placement is **explicitly** specified by the developer. Even for multitier solutions with automatic placement, the assignment to different hosts is an integral part of the programming model. For example, specific parts of the code have a fixed placement (e.g., interaction with the web browser's DOM must be on the client) or the developer is given the ability to use location annotations to enforce a certain placement.

The code that is assigned to different places is either (1) **partitioned** (statically or dynamically) into different programs or (2) separated into different **stages**, where the execution of one stage generates the next stage and can inject values computed in the current stage into the next one. When accessing a value of another partition in approach (1), the value is looked up remotely over the network and the local program continues execution with the remote value after receiving it. For handling remote communication asynchronously, remote accesses are either compiled to continuation-passing style or asynchronicity is exposed to the developer using local proxy objects such as futures [Baker and Hewitt 1977]. Using approach (2) for web applications, the server stage runs and creates the program to be sent to the client. When generating the client program, references to server-side values are spliced into client code, i.e., the client program that is sent already contains the injected server-side values. Such staged execution reduces communication overhead since server-side values accessed by the client are part of the generated client program.

In the case of web applications, as response to an HTTP request, the server delivers the program to the client which executes it in the browser. Upon executing, the client program can connect to the server for additional communication. For

multitier languages that do not target web applications, the split programs start independently on different hosts and connect to other parts upon execution, e.g., using peer-to-peer service discovery in AmbientTalk/R [Carreton et al. 2010].

We first consider placement based on the different functionalities of the application logic which naturally belong to different tiers. Then we present approaches where there are multiple options for placement and the multitier programming framework assigns functionalities to tiers based on various criteria such as performance optimization and privacy.

Placement based on functional properties In most multitier languages, the placement of each functionality is fully defined by the programmer by using an escaping/quoting mechanism (Hop [Serrano et al. 2006], Ur/Web [Chlipala 2015], Eliom [Radanne et al. 2016]), annotations (Links [Cooper et al. 2006]) or a type-level encoding (ML5 [Murphy et al. 2007], Gavial [Reynders et al. 2020], ScalaLoci [Weisenburger et al. 2018a]). Placement allows separate parts of the multitier program to execute on different hosts. The compile-time separation into different components either relies on (whole-)program analysis (Ur/Web, ML5) or supports modular separation (Eliom, ScalaLoci), where each module can be individually split into multiple tiers. On the other hand, dynamic separation is performed at run time (Links, Hop).

When the placement specification is incomplete, there is room for alternative placement choices, in which case slicing [Weiser 1981] detects the dependencies between the fragments which are manually assigned by developers and the rest of the code base, ultimately determining the splitting border. For example, in StiP.js [Philips et al. 2018], code fragments are assigned to a tier based on annotations, then slicing uncovers the dependencies. This solution allows developing multitier web applications in existing general-purpose languages as well as retaining compatibility with development tools. In the slicing process, placement can be constrained not only explicitly, but also based on values' behavior, e.g., inferring code locations using control flow analysis or rely on elements for which the location is known [Chong et al. 2007b; Philips et al. 2018; Rajchenbach-Teller and Sinot 2010] (e.g., database access takes place on the server, interaction with the DOM takes place on the client). This complicates the integration into an existing language, especially in presence of effects, and is less precise than explicit annotations – hindering, e.g., the definition of data structures that combine fragments of client code and other data [Radanne et al. 2016].

Placement strategies For the functionalities that can execute on the client and on the server, multitier approaches either place unannotated code both on the client

and on the server (e.g., Links [Cooper et al. 2006], Opa [Rajchenbach-Teller and Sinot 2010], ScalaLoci [Weisenburger et al. 2018a]) or compute the placement that minimizes the communication cost between tiers (e.g., Distributed Orc [Thywissen et al. 2016]). Neubauer and Thiemann [2005] allow a propagation strategy to produce different balances for the amount of logic that is kept on the client and on the server, starting the propagation from some predefined operators whose placement is fixed. The propagation strategy uses a static analysis based on location preferences and communication requirements to optimize performance (contrarily to many multitier approaches where the choice is left to the programmer). Jif/split [Zdancewic et al. 2002] considers placement based on security concerns: Protection of data confidentiality is the principle to guide the splitting. The input is a program with security annotations and a set of trust declarations to satisfy. The distributed output program satisfies all security policies. As a result, programmers can write code that is agnostic to distribution but features strong guarantees on information flow. Similarly, Swift [Chong et al. 2007a] also partitions programs based on security labels, but focuses on the Web domain where the trust model assumes a trusted server that interacts with untrusted clients.

An exception to the approaches above – which all adopt a compile time splitting strategy – is Fission [Guha et al. 2017], which uses information flow control to separate client and server tiers at run time. The dynamic approach allows supporting JavaScript features that are hard to reason about statically, such as `eval`, as well as retaining better compatibility with tooling.

2.6.3 Placement Specification and Granularity

Placement specification in multitier languages is defined in various ways and at different granularity levels. Multitier languages follow several approaches to specify the execution location, allowing the composition of code belonging to different hosts in the same compilation unit. Table 2.4 on the next page classifies the multitier languages based on the placement specification approach and the granularity given in the first row. For example, Hop.js allows escaping arbitrary expressions to delimit code of a different tier. Links uses annotations on top-level bindings to specify the tier to which a binding belongs.

Placement specification We identified the following strategies used by multitier languages to determine placement:

Dedicated tier assignment always associates certain language constructs to a tier, e.g., top-level name bindings are always placed on the server or every class represents a different tier.

Table 2.4 Placement approach.

Language	Placement Specification Approach for given Granularity						
	Expression	Binding	Block	Top-Level Binding	Top-Level Block	Class/Module	File
Hop/Hop.js	escaping/quoting	·	·	annotation	·	·	·
Links	·	·	·	annotation	·	·	·
Opa	·	annotation and static analysis	·	·	·	·	·
StiP.js	·	·	·	·	annotation and static analysis	·	·
Ur/Web	escaping/quoting	·	·	dedicated	·	·	·
Eliom/Ocsigen	escaping/quoting	·	·	annotation	·	annotation	·
Gavial	type	·	·	·	·	·	·
AmbientTalk/R	·	·	·	·	·	dedicated	·
ML5	type	·	·	·	·	·	·
ScalaLoci	type	·	·	type	·	·	·
WebSharper	·	·	·	annotation	·	annotation	·
Haste	type	·	·	·	·	·	·
Fun	·	·	·	dedicated	·	·	·
Koka	·	·	·	type	·	·	·
Multi-Tier Calculus	static analysis	·	·	·	·	·	·
Swift	static analysis	·	·	·	·	·	·
Volta	·	·	·	·	·	annotation	·
J-Orchestra	·	·	·	·	·	external	·
Meteor	·	·	dynamic run time check	·	·	·	directory
GWT	·	·	·	·	·	·	directory

Annotations specify the tier to which the annotated code block or definition belongs, driving the splitting process.

Escaping/quoting mechanisms are used when the surrounding program is placed on a specific tier, e.g., the server, and nested expressions are escaped/quoted to delimit the parts of the code that run on another specific tier, e.g., the client.

Types of expressions or name bindings determine the tier, making placement part of the type system and amenable to type checking.

Static analysis determines the tier assignment at compile time based on functional properties of the code (such as access to a database or access to the DOM of the webpage).

Dynamic run time checks allow developers to check at run time which tier is currently executing the running code, and select the tier-specific behavior based on such condition.

The following strategies are used by approaches lacking language-level support for placement:

External configuration files assign different parts of the code (such as classes) to different tiers.

Different directories are used to distinguish among the files containing the code for different tiers.

Links [Cooper et al. 2006] and Opa [Rajchenbach-Teller and Sinot 2010] provide dedicated syntax for placement (e.g., `fun f() client` and `fun f() server` in Links). Volta [Manolescu et al. 2008] relies on the C# base language's custom attribute annotations to indicate the placement of abstractions (e.g., `[RunAt("Client")] class C`). WebSharper [Bjornson et al. 2010] uses a `JavaScript` F# custom attribute to instruct the compiler to translate a .NET assembly, a module, a class or a class member to JavaScript (e.g., `[<JavaScript>] let a = ...`). Stip.js [Philips et al. 2018] interprets special forms of comments (e.g., `/* @client */ {...}` and `/* @server */ {...}`). While multitier languages usually tie the placement specification closely to the code and define it in the same source file, approaches like J-Orchestra [Tilevich and Smaragdakis 2002] require programmers to assign classes to the client and server sites in an XML configuration file.

ML5 [Murphy et al. 2007] captures the placement explicitly in the type of an expression. For example, an expression *`expr`* of type `string @ server` can be executed on the `home` world using `from server get` *`expr`*. The placement of every expression is determined by its type and the compiler ensures type-safe composition of remote expressions through `from ... get`. Similarly, in ScalaLoci [Weisenburger et al. 2018a], a binding *`value`* of type `String on Server` can be accessed remotely using `value.asLocal`. Haste [Ekblad and Claessen 2014] also features a type-based placement specification using monadic computations by wrapping client and server code into different monads. Koka Leijen 2014 uses a type and effect system to capture which functions can only be executed on the client and which functions can only be executed on the server, preventing cross-tier access without explicitly sending and receiving messages.

Placement granularity On a different axis, existing multitier approaches cover a wide granularity spectrum regarding the abstractions for which programmers can define placement: **files** (e.g., GWT [Kereki 2010]), **classes** (e.g., Volta [Manolescu et al. 2008], J-Orchestra [Tilevich and Smaragdakis 2002]), **top-level code blocks** (e.g., Stip.js [Philips et al. 2018]), **top-level bindings** (e.g., Links [Cooper et al. 2006]), (potentially nested) **blocks** (e.g., Meteor [Strack 2012]), **bindings** (e.g., Opa [Rajchenbach-Teller and Sinot 2010]) and **expressions** (e.g., Eliom [Radanne et al. 2016], ML5 [Murphy et al. 2007]). Specification granularities supported by a language are not mutually exclusive, e.g., ScalaLoci [Weisenburger et al. 2018a] supports placed top-level bindings and nested remote blocks. In Hop [Serrano et al. 2006] and Ur/Web [Chlipala 2015], which target web applications where

the execution of server code is triggered by an HTTP client request, all top-level bindings define server-side code and nested client-side code is escaped/quoted at the granularity of expressions. Eliom [Radanne et al. 2016] supports both nested client expressions and annotated top-level client/server bindings.

The approach most akin to traditional languages is to force programmers to define functionalities that belong to different hosts in separated compilation units, such as different Java packages (GWT [Kereki 2010]) or different directories (Meteor [Strack 2012]) or dynamically check the tier on which the code is currently executed (Meteor). An even coarser granularity is distribution at the software component level. R-OSGi [Rellermeyer et al. 2007] is an OSGi extension where developers specify the location of remote component loading and Coign [Hunt and Scott 1999] extends COM to automatically partition and distribute binary applications. These solutions, however, significantly depart from the language-based approach of multitier programming.

2.6.4 Communication Abstractions

Multitier approaches provide dedicated abstractions intended to simplify implementing remote communication, which differ considerably among languages. Table 2.5 on the facing page provides an overview over these abstractions. We identified web page requests, remote procedure calls, publish–subscribe schemes, reactive programming and shared state as the main communication mechanisms used by multitier languages. Languages either support specific forms of communication only in a single direction – either from client to server or from server to client – or support bidirectional communication (potentially requiring the client to initiate the communication). multitier languages also differ in whether they make remote communication explicit (and with it, the associated performance impact) or completely transparent to the developer.

Remote communication mechanisms are either integrated into the language using convenient syntactic constructs (e.g., `from ... get` *`expr`* in ML5 [Murphy et al. 2007], *`value`*`.asLocal` in ScalaLoci [Weisenburger et al. 2018a] or `rpc` *`fun`* in Ur/Web [Chlipala 2015]), or are made available through a set of standard library functions that come with the language (e.g., `webSocket.send(`*`message`*`)` in Hop.js [Serrano and Prunet 2016] or `service.`*`fun`*`(new AsyncCallback() {...})` in GWT [Kereki 2010] or `Meteor.call("`*`fun`*`", function(error, result) {...})` in Meteor [Strack 2012]). We list the communication approaches found in the respective multitier languages in Table 2.5 on the next page. Developers can, however, implement such communication mechanisms that are not supported out-of-the-box (by dedicated language features or as part of the standard library) as an external

Table 2.5 Communication abstractions.

Language	Communication Abstraction				
	Remote Procedures	Message Passing	Publish–Subscribe	Reactive Programming	Shared State
Hop/Hop.js	●$^{c\to s}$	○	○	·	·
Links	●^{t,c}	●	·	·	·
Opa	●^{t}	●	·	·	·
StiP.js	●^{t}	·	●	·	●
Ur/Web	●$^{c\to s}$	●$^{s\to c}$	·	·	·
Eliom/Ocsigen	○	○	·	○	·
Gavial	·	·	·	●	·
AmbientTalk/R	·	·	●	●	·
ML5	●	·	·	·	·
ScalaLoci	●	·	·	●	·
WebSharper	●	·	·	·	·
Haste	●$^{c\to s}$	·	·	·	·
Fun	·	·	·	·	●
Koka	·	●	·	·	·
Multi-Tier Calculus	●^{t}	·	·	·	·
Swift	●^{t}	·	·	·	·
Volta	●^{t}	·	·	·	·
J-Orchestra	●^{t}	·	·	·	·
Meteor	○	·	○	·	○
GWT	○	·	·	·	·

● Language support
○ Support through libraries
$^{c\to s}$ From client to server only
$^{s\to c}$ From server to client only
^{t} Fully transparent remote procedure
^{c} Client-initiated

library, e.g., providing a library that supports event-based communication based on remote procedure calls or using a persistent server to emulate shared data structures. We do not consider such external solutions here. We identify the following remote communication mechanisms:

Remote procedures are the predominant communication mechanism among multitier languages. Remote procedures can be called in a way similar to local functions – either completely transparently or using a dedicated remote invocation syntax – providing a layer of abstraction over the network between the call site and the invoked code.

Message passing abstractions are closer to the communication model of the underlying network protocols, where messages are sent from one host to another.

Publish–subscribe allows tiers to subscribe to topics of their interest and receive the messages published by other tiers for those topics.

Reactive programming for remote communication defines data flows across tiers through event streams or time-changing values that, upon each change, automatically update the derived reactive values on the remote tiers.

Shared state makes any updates to a shared data structure performed on one tier available to other tiers accessing the data structure.

Multitier languages that target the Web domain follow a traditional request–response scheme, where web pages are generated for each client request and the

client interacts with the server by user navigation. Both Hop [Serrano et al. 2006] and Eliom [Radanne et al. 2016] allow client and server expressions to be mixed. All server expressions are evaluated on the server before delivering the web page to the client and client expressions are evaluated on the client. Only a single communication takes place at this point. Hop additionally provides traditional client–server communication via asynchronous callbacks, whereas Eliom supports more high-level communication mechanisms based on reactive programming through libraries.

WebDSL [Groenewegen et al. 2008], for example, is an external DSL for web applications to specify the data model and the pages to view and edit data model objects. HTML code is generated for pages, which is reconstructed upon every client request.

Call-based communication Multitier languages provide communication abstractions for client–server interaction not necessarily related to page loading, including RPC-like calls to remote functions, shared state manipulation or message passing. Abstracting over calling server-side services and retaining the result via a local callback, Links [Cooper et al. 2006] allows bidirectional remote function calls between client and server. RPC calls in Links, however, hide remote communication concerns completely, which has been criticized because the higher latency is not explicit [Kendall et al. 1994]. In contrast, Links' more recent message passing communication mechanism features explicit send and receive operations.

In both Ur/Web [Chlipala 2015] and Opa [Rajchenbach-Teller and Sinot 2010], server and client can communicate via RPCs or message-passing channels. Due to the asymmetric nature of client–server web applications, Ur/Web follows a more traditional approach based on RPCs for client-to-server communication and provides channels for server-to-client communication.

Event-based communication Some languages adopt event propagation either in a publish–subscribe style or combined with abstractions from reactive programming. AmbientTalk [Dedecker et al. 2006] uses a publish–subscribe middleware in the context of loosely coupled mobile devices where no stable data flow can be assumed. Hiphop [Berry et al. 2011], which extends Hop [Serrano et al. 2006] with synchronous data flows, borrows ideas from synchronous data flow languages à la Esterel [Berry and Gonthier 1992]. The approach provides substantial guarantees on time and memory bounds, at the cost, however, of significantly restricting expressivity. In ScalaLoci [Weisenburger et al. 2018a], Gavial [Reynders et al. 2020], AmbientTalk/R [Carreton et al. 2010] or libraries for Eliom [Radanne et al. 2016],

tiers expose behaviors (a.k.a. signals) and events in the style of functional reactive programming to each other.

Distributed shared state Meteor [Strack 2012] provides *collections* to store JSON-like documents and automatically propagate changes to the other tier. Similarly, in Fun [Westin 2010], a language for real-time web applications, modifications to variables bound to the `Global` object are automatically synchronized across clients, thus avoiding the need for manual remote state updates. Multitier languages usually support (or even require) a central server component, enabling shared state via the server as central coordinator that exposes its state to the clients.

2.6.5 Formalization of Multitier Languages

From a formal perspective, multitier programming has been investigated in various publications. In this section, we first present a classification of existing formal models that have been explored using three analysis directions: the formalization approach, the proof methods and the properties considered in the formalization. Finally, we describe the formalizations of multitier languages in more details, classifying them according to the points above.

Techniques and scope Existing formal models for multitier languages that specify an operational *semantics* follow three main approaches: (s1) they formalize how a single **coherent multitier program** is executed modeling how computation and communication happen in the whole distributed setting (e.g., with a semantics where terms can be reduced at different locations) [Boudol et al. 2012; Neubauer and Thiemann 2005; Radanne et al. 2016; Weisenburger et al. 2018a], (s2) they specify a **splitting transformation** that describes how tier-specific programs are extracted from multitier code and they provide an independent reduction model for the split tiers [Cooper and Wadler 2009; Neubauer and Thiemann 2005; Radanne et al. 2016] or (s3) they specify the semantics in terms of an **existing calculus** [Leijen 2014], i.e., the semantics of a calculus not specific to multitier languages is reinterpreted for multitier programming, e.g., different effects in a type and effect system represent different tiers. The continuation-based denotational semantics by Serrano and Queinnec [2010] is an exception to the operational approach. It focuses on a sequential fragment of Hop to model dynamic server-side client code generation.

Based on the models above, researchers looked at *properties* including (p1) **type soundness** as progress and preservation [Boudol et al. 2012; Leijen 2014; Neubauer and Thiemann 2005; Weisenburger et al. 2018a], (p2) **behavioral equivalence** of the execution of the source multitier program (cf. a1) and the interacting concurrent

Table 2.6 Formalization approach.

Language	Proved Properties			
	Type Soundness of Coherent multitier Program	Type Soundness based on Existing Calculus	Behavioral Equivalence of Splitting Transformation	Domain-Specific Properties
Hop/Hop.js	denotational	·	·	operational (same-origin policy)
Links	·	·	operational	·
Eliom/Ocsigen	operational	·	operational	·
ScalaLoci	operational	·	·	·
Multi-Tier Calculus	operational	·	operational	·
Koka	·	operational	·	·

execution of the tier-specific programs (cf. a2) [Cooper and Wadler 2009; Neubauer and Thiemann 2005; Radanne et al. 2016], and (p3) **domain-specific properties** that are significant in a certain context, such as secure compilation [Baltopoulos and Gordon 2009] or performance for data access [Chandra et al. 2008] as well as domain-specific properties, such as host reachability in software defined networks [Nelson et al. 2014]. Crucially, the fact that multitier languages model client and server together enables reasoning about global data flow properties such as privacy. The small-step semantics of Hop [Boudol et al. 2012] has been used to model the browser's same-origin policy and define a type system that enforces it. A similar approach has been proposed to automatically prevent code injection for web applications [Luo et al. 2011]. Splitting in Swift [Chong et al. 2007a] is guaranteed to keep server-side private information unreachable by client-side programs.

Researchers adopted *proof methods* that belong to two categories: (m1) perform the proofs directly on the semantics that describes the whole system and/or the splitting transformation [Boudol et al. 2012; Cooper and Wadler 2009; Neubauer and Thiemann 2005; Radanne et al. 2016; Weisenburger et al. 2018a] or (m2) leverage proved properties of an existing calculus [Cooper et al. 2006; Leijen 2014].

Formalizations Table 2.6 provides a classification of the formalizations of multitier languages. For the discussion, we leave out languages lacking a formal development. Most formalizations model multitier applications as single coherent programs, providing soundness proofs for the multitier language. Another common approach for reasoning about the behavior of multitier code is to formally define the splitting transformation that separates multitier code into its tier-specific parts to show behavioral equivalence of the original multitier program and the split programs after the transformation. In the case of Hop [Boudol et al. 2012] formal reasoning focuses on properties specific to the Web domain, e.g., conformance of multitier programs to the browser's same-origin policy. Koka's effect system [Leijen 2014] can be used to implement different tiers in the same compilation unit. The

sound separation into different tiers in Koka follows from the soundness of the effect system.

The seminal work by Neubauer and Thiemann [2005] presents a multitier calculus for web applications. A static analysis on a simply-typed call-by-value lambda calculus determines which expressions belong to each location and produces the assignment of the code to the locations, which results in a lambda calculus with annotated locations. A further translation to a multitier calculus (s1) explicitly models opening and closing of communication channels. Type soundness for the multitier calculus is proved (p1). The splitting transformation (s2), which extracts a program slice for each location, is proved to generate only valid programs wrt. the source (p2). The transformed program is considered valid if it is weakly bisimilar [Park 1981] to the source program, i.e, if it performs the same operations with the same side effects and the operations are in the same order (m1).

Boudol et al. [2012] provide a small-step operational semantics for Hop, which covers server-side and client-side computations, concurrent evaluation of requests on the server and DOM manipulation (s1). For Hop, based on Scheme, which does not feature a static type system, the authors define a type system for "request-safety" (p1), which ensures that client code will never request server-side services that do not exist. Request-safety is proven sound (m1).

The formalization of the Links programming language [Cooper et al. 2006] is based on RPC calculus [Cooper and Wadler 2009] (m2) – an extension of lambda calculus – which models location awareness for stateful clients and stateless servers. The RPC calculus is transformed (s2) into a client program and a server program in the client/server calculus. The transformation is proved to be correct and complete (m1). Further, a location-aware calculus, which is the theoretical foundation for the Links programming language, and a translation to RPC calculus is provided (p2). A simulation that proves that the behavior of the transformed program in the client/server calculus conforms to the behavior of the source program in location-aware calculus is left to future work.

Eliom [Radanne et al. 2016] is formalized as a multitier extension of core ML. The authors provide an operational semantics that formalizes the execution for an Eliom program (s1) and provide a translation (s2) separating an Eliom program into server and client ML programs. Besides subject reduction (p1), the authors prove the equivalence of the high level multitier semantics with the semantics of the compiled client and server languages after splitting by simulation (p2). The simulation shows that, for any given source program, every reduction can be replayed in the transformed programs (m1). Eliom separates type universes for client and server, allowing the type system to track which values belong to which

Table 2.7 Distribution topologies.

Language	Distribution Topology			
	Client–Server	Client–Server + Database	Peer-to-Peer	Specifiable
Hop/Hop.js	●	·	·	·
Links	·	●	○ [1]	·
Opa	·	●	·	·
StiP.js	●	·	·	·
Ur/Web	·	●	·	·
Eliom/Ocsigen	●	·	·	·
Gavial	●	·	·	·
AmbientTalk/R	·	·	●	·
ML5	●	·	·	○
ScalaLoci	·	·	·	●
WebSharper	●	·	·	·
Haste	●	·	·	·
Fun	●	·	·	·
Koka	●	·	·	○
Multi-Tier Calculus	●	·	·	○
Swift	●	·	·	·
Volta	●	·	·	·
J-Orchestra	·	·	·	●
Meteor	●	·	·	·
GWT	●	·	·	·

● Supported
○ Support conceptually possible, but not supported by the provided examples or the implementation
1 Client-to-client communication transparently through central server

side. Eliom, however, leaves out interactive behavior, formalizing only the creation of a single page.

In ScalaLoci's formal semantics [Weisenburger et al. 2018a], the reduction relation is labeled with the distributed components on which a term is reduced (s1). The formalization models the peers of the distributed system and interactive remote access between peers, providing soundness properties for the encoding of placement at the type level, e.g., that terms are reduced on the instances of the peers on which they are placed (p1). The type system is proven sound (m1).

Using the Koka language, it is possible to define a splitting function for the server and client parts of a program [Leijen 2014] based on Koka's ability to separate effectful computations (s3), which guarantees type soundness for the split programs (p1), e.g., an application can define a *client* effect consisting of DOM accesses and a *server* effect consisting of I/O operations (m2).

2.6.6 Distribution Topologies

Table 2.7 gives an overview over the distribution topologies supported by multitier languages. The majority of multitier approaches specifically targets **client–server** applications in the Web domain. Besides the client and the server tier, Links [Cooper et al. 2006], Opa [Rajchenbach-Teller and Sinot 2010] and Ur/Web [Chlipala 2015] also include language-level support for the **database** tier. Other multitier languages

require the use of additional libraries to access a database (e.g., Hop [Serrano et al. 2006] or Eliom [Radanne et al. 2016]).

Only few approaches target other distribution topologies: AmbientTalk [Dedecker et al. 2006] focuses on mobile ad hoc networks and allows services to be exported and discovered in a **peer-to-peer** manner, where peers are loosely coupled. ML5 [Murphy et al. 2007] is a multitier language which adopts the idea of *possible worlds* from models of modal logic to represent the different tiers in the distributed system. Worlds are used to assign resources to different tiers. Although this approach is potentially more general than the client–server model, allowing for the definition of different tiers, the current compiler and runtime target web applications only. Similarly, in the multitier calculus by Neubauer and Thiemann [2005], locations are members of a set of location names that is not restricted to client and server. Their work, however, focuses on splitting code between a client and as server. Session-typed channels in Links [Cooper et al. 2006] provide the illusion of client-to-client communication, but messages are routed through the server. In J-Orchestra [Tilevich and Smaragdakis 2002], developers can define different interconnected network sites in a configuration file.

ScalaLoci [Weisenburger et al. 2018a] allows developers to **specify a distributed system's topology** by declaring types representing the different components and their relation. Thus, developers can define custom architectural schemes (i.e., not only client–server) and specify various computing models (e.g., pipelines, rings, or master–worker schemes).

2.7 Related Approaches

In this section, we provide an overview of related research areas that influenced research on multitier programming and the design of our ScalaLoci multitier language or share concepts with the multitier paradigm.

Programming languages and calculi for distributed systems Multitier programming belongs to a long tradition of programming language design for distributed systems with influential distributed languages like Argus [Liskov 1988], Emerald [Black et al. 2007], Distributed Oz [Haridi et al. 1997; Van Roy et al. 1997], Dist-Orc [AlTurki and Meseguer 2010] and Jolie [Montesi et al. 2014]. A number of frameworks for big data computations such as Flink [Carbone et al. 2015], Spark [Zaharia et al. 2012], Gearpump [Zhong et al. 2014], Dryad [Isard and Yu 2009], PigLatin [Olston et al. 2008] and FlumeJava [Chambers et al. 2010], have been proposed over the last few years, motivated by refinements and generalizations of

the original MapReduce [Dean and Ghemawat 2008] model. In these frameworks, fault tolerance, replication and task distribution are handled transparently by the runtime. More recently, there have been contributions to specific aspects in the design of programming languages that concern the support for distributed systems, such as cloud types to ensure eventual consistency [Burckhardt et al. 2012], conflict-free replicated data types (CRDT) [Shapiro et al. 2011b], language support for safe distribution of computations [Miller et al. 2014] and fault tolerance [Miller et al. 2016], as well as programming frameworks for mixed IoT/Cloud development, such as Ericsson's Calvin [Persson and Angelsmark 2015].

Several formal calculi model distributed systems and abstract, to various degrees, over placement and remote communication. The Ambient calculus [Cardelli and Gordon 1998] models concurrent systems with both mobile devices and mobile computation. In Ambient, it is possible to define named, bounded places where computations occur. Ambients can be moved to other places and are nested to model administrative domains and their access control. The Join calculus [Fournet and Gonthier 1996] defines processes that communicate by asynchronous message passing over channels in a way that models an implementation without expensive global consensus. CPL [Bračevac et al. 2016] is a core calculus for combining services in the cloud computing environment. CPL is event-based and provides combinators that allow safe composition of cloud applications.

Choreographies In choreographic programming, a concurrent system is defined as a single compilation unit called *choreography*, which is a global description of the interactions and computations of a distributed system's connected components [Lanese et al. 2008; Montesi 2014; W3C WS-CDL Working Group 2005]. Similar to multitier programming, the compiler automatically produces a correct implementation for each component, e.g., as a process or as a microservice [Carbone and Montesi 2013]. While multitier languages abstract over communication, choreographic programming is *communication-centric* andle the expected communication flow among components is defined explicitly. The compiler is responsible for generating code that strictly abides by this flow. Choreographic programming's formal foundations are rooted in process calculi [Baeten 2005]. It has been used to investigate new techniques on information flow control [Lluch Lafuente et al. 2015], deadlock-free distributed algorithms [Cruz-Filipe and Montesi 2016] and protocols for dynamic run time code updates for components [Preda et al. 2017]. Role parameters in the choreographic language Choral [Giallorenzo et al. 2020] recall ScalaLoci's abstract peer types [Weisenburger and Salvaneschi 2019a]: They can be freely instantiated with different arguments, further allowing for components to dynamically switch the roles in the distributed system at run time.

Actor model The *actor model*, initially described by Hewitt [Hewitt et al. 1973] and available in popular implementations such as Erlang OTP [Armstrong 2010] and Akka [Lightbend 2009a], encapsulates control and state into computation units that run concurrently and exchange messages asynchronously [Agha 1986]. The decoupling offered by asynchronous communication and by the no-shared-memory approach enables implementing scalable and fault-tolerant systems. De Koster et al. [2016] classify actor systems into four different variants: (i) the *classic actor model* allows for changing the current interface of an actor (i.e., the messages which an actor can process) by switching between different named behaviors, which handle different types of messages, (e.g., Rosette [Tomlinson et al. 1988], Akka [Lightbend 2009a]), (ii) *active objects* define a single entry point with a fixed interface (e.g., SALSA [Varela and Agha 2001], Orleans [Bykov et al. 2011]), (iii) *process-based actors* are executed once and run until completion, supporting explicit receive operations during run time (e.g., Erlang [Armstrong 2010], Scala Actor Library [Haller and Odersky 2009]) and (iv) *communicating event-loops* combine an object heap, a message queue and an event loop and support multiple interfaces simultaneously by defining different objects sharing the same message queue and event loop (e.g., E [Miller et al. 2005]). Actors, however, are a relatively low-level mechanism to program distributed systems, leaving programmers the manual work of breaking applications between message senders and message handlers. The survey by Boer et al. [2017] provides an overview of the current state of research on actors and active object languages.

Reactive programming Event-based applications are traditionally implemented using the Observer design pattern, which comes with several issues including inversion of the control flow, inducing side-effecting operations to change the state of the application [Cooper and Krishnamurthi 2006]. Reactive programming overcomes the problems of the Observer pattern by allowing for data flows to be defined directly and in a more declarative manner. When declaring a reactive value, the reactive system keeps track of all dependencies of the reactive value and updates it whenever one of its dependencies changes. Reactives allow for better maintainability and composability as compared to observers [Maier et al. 2010] and lead to code that is more composable, more compact [Salvaneschi and Mezini 2014] and easier to understand [Meyerovich et al. 2009; Salvaneschi et al. 2014a].

Functional reactive programming was originally proposed by Elliott and Hudak [1997] to declaratively program visual animations. Formally modeling continuous time led to a denotational semantics where time-changing variables are functions from time to values [Nilsson et al. 2002]. Functional reactive programming has since been applied to a number of fields including robotics [Hudak et al. 2003],

network switches [Foster et al. 2011] and wireless sensor networks [Newton et al. 2007]. User interfaces have become a largely popular application field for reactive programming. Flapjax [Meyerovich et al. 2009] pioneered using reactive programming in JavaScript web clients. Elm [Czaplicki and Chong 2013], a functional language akin to Haskell that compiles to JavaScript, adopts a similar approach. Flapjax provides behaviors and event streams, while Elm uses only signals to model both time-varying values and events. Microsoft Reactive Extensions (Rx) [Meijer 2010] offer abstractions for event streams. Rx is available for both Java (RxJava) and JavaScript (RxJS), but as separate implementations, i.e., reactive dependencies cannot cross the boundaries among different hosts.

Other recent research directions regarding reactive programming include debugging [Banken et al. 2018; Perez and Nilsson 2017; Salvaneschi and Mezini 2016], thread-safe concurrency [Drechsler et al. 2018], and application to new domains such as IoT and edge computing [Calus et al. 2017] and autonomous vehicles [Finkbeiner et al. 2017]. In the distributed setting, the DREAM reactive middleware [Margara and Salvaneschi 2018, 2014] allows selecting different levels of consistency guarantees for distributed reactive programming. CRDTs have been used to handle state replication for signals shared among hosts [Myter et al. 2016]. Mogk et al. [2018a, 2019] extend the reactive programming language REScala [Salvaneschi et al. 2014b] with distributed fault handling based on state replication via CRDTs and state snapshotting.

Some multitier languages adopt reactive programming features. Ur/Web [Chlipala 2015] supports functional reactive user interfaces on the client. It, however, does not directly support data flows over multiple hosts. Communication from the server to the client is achieved by message-passing channels and from client to server by remote procedure calls. Eliom [Radanne et al. 2016] and Gavial [Reynders et al. 2020] provide signals and events – both can propagate values remotely. Yet, they only support client–server web applications. AmbientTalk/R [Carreton et al. 2010] targets mobile applications with loosely coupled devices and no stable data flow. It provides reactives on top of a publish–subscribe middleware. In contrast to other multitier languages, it does not specifically support the client–server architecture. Similar to the other approaches, however, it is restricted to a predefined architectural model and it does not supported more complex architectures, e.g., with master and worker components etc.

PGAS languages Partitioned global address space languages (PGAS) [De Wael et al. 2015] provide a high-level programming model for high-performance parallel execution. For example, X10 [Charles et al. 2005] parallelizes task execution based on a work-stealing scheduler, enabling programmers to write highly scalable code.

Its programming model features explicit fork/join operations to make the cost of communication explicit. X10's sophisticated dependent type system [Chandra et al. 2008] captures the *place* (the heap partition) a reference points to. Similar to multitier languages, PGAS languages aim at reducing the boundaries between hosts, adopting a shared global address space to simplify development. The scope of PGAS languages, however, is very diverse – they focus on high performance computing in a dedicated cluster, while multitier programming targets client–server architectures on the Internet.

Software architectures Software architectures [Garlan and Shaw 1994; Perry and Wolf 1992] organize software systems into components, specifying their connections as well as constraints on their interaction. Architecture description languages (ADL) [Medvidovic and Taylor 2000] provide a mechanism for high-level specification and analysis of large software systems to guide, for example, architecture evolution. ADLs define components, connectors, architectural invariants and a mapping of architectural models to an implementation infrastructure. Yet, ADLs are often detached from implementation languages. ArchJava [Aldrich et al. 2002] paved the way for consolidating architecture specification and implementation in a single language. However, ArchJava does not specifically address distributed systems nor multitier programming. Some approaches are at the intersection of multitier and modeling languages: Hilda [Yang et al. 2007] is a web development environment for data-driven applications based on a high-level declarative language similar to UML, which automatically partitions multitier software.

Influenced by ADLs and object-oriented design, component models [Crnkovic et al. 2011] provide techniques and technologies to build software systems starting from units of composition with contractually specified interfaces and dependencies which can be deployed independently [Szyperski 2002]. Component-based development (CBD) aims at separating different concerns throughout the whole software system, defining component interfaces for interaction with other components and mechanisms for composing components, providing strong interfaces to other modules. In a distributed setting, CBD usually models the different components of the distributed system as separate components, forcing developers to modularize along network boundaries.

Module systems Rossberg and Dreyer [2013] design MixML, a module system that supports higher-order modules and modules as first-class values and combines ML modules' hierarchical composition with mixin's recursive linking of separately compiled components. The *concept pattern* [Oliveira et al. 2010] uses Scala's implicit resolution to enable retroactive extensibility in the style of Hakell's type

classes. The Genus programming language provides modularization abstractions that support generic programming and retroactive extension in the object-oriented setting in a way similar to the concept pattern [Zhang et al. 2015; Zhang and Myers 2017]. Family polymorphism explores definition and composition of module hierarchies. The J& language supports hierarchical composability for nested classes in a mixin fashion [Nystrom et al. 2004, 2006]. Nested classes are also supported by Newspeak, a dynamically typed object-oriented language [Bracha et al. 2010]. Virtual classes [Ernst et al. 2006] enable large-scale program composition through family polymorphism. Dependent classes [Gasiunas et al. 2007] generalize virtual classes to multiple dispatching, i.e., class constructors are dynamically dispatched and are multimethods.

Metaprogramming Compile-time metaprogramming was pioneered by the Lisp macro system [Hart 1963] that supports transformations of arbitrary Lisp syntax, facilitating the addition of new syntactic forms. Racket, a Lisp dialect, is designed to allow building new languages based on macros [Felleisen et al. 2015]. In Template Haskell [Sheard and Jones 2002], Haskell metaprograms generate ASTs, which the compiler splices into the call sites. Such metaprograms do not take ASTs as input without explicitly using quasiquotation at the call site. Template Haskell has been used to optimize embedded domain-specific languages at compile time [Seefried et al. 2004]. Rust supports hygienic *declarative macros* that expand before type-checking and define rewrite rules to transform programs based on syntactic patterns. Rust's *procedural macros* are more powerful using Rust code to rewrite token streams and accessing compiler APIs. A combination of Rust's type system and macro system was shown to support a shallow embedding of the lambda calculus [Headley 2018].

Multi-stage programming Multi-stage programming splits program compilation into a number of stages, where the execution of one stage generates the code that is executed in the next stage. MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003] provide a quasi-quotation mechanism that is statically scoped to separate stages syntactically. Quoted expressions are not evaluated immediately but they generate code to be executed in the next stage. The Hop [Serrano et al. 2006] multitier language uses multi-stage programming to construct client code at the server side. Instead of using syntactic quotations, lightweight modular staging [Rompf and Odersky 2010] employs a staging approach based on types, combining staged code fragments in a semantic way with strong guarantees on well-formedness and type soundness. Using lightweight modular staging with the Scala-Virtualized [Moors et al. 2012] modified Scala compiler also enables overloading Scala language constructs such as loops and control structures.

Language integration for database queries Properly integrating query languages into general-purpose languages is a long-standing research problem [Atkinson and Buneman 1987]. Compiling embedded queries into SQL was pioneered by the Kleisli system [Wong 2000]. LINQ [Torgersen 2007] is a language extension based on Kleisli's query compilation technique to uniformly access different data sources such as collections and relational databases. The Links [Cooper et al. 2006] multitier language also relies on this technique for providing access to the database tier. Recent approaches for embedding database queries, such as JReq [Iu et al. 2010], Ferry [Grust et al. 2009], DBPL [Schmidt and Matthes 1994], Slick [Lightbend 2014] or Quill [Ioffe 2015] also follow a functional approach without object-relational mapping.

Domain-specific languages Several survey papers are available in the literature that provide an extensive overview of DSLs [Deursen and Klint 1998; Deursen et al. 2000; Mernik et al. 2005; Spinellis 2001]. Wile [2004] provides a compendium of lessons learned on developing domain-specific languages providing empirically derived guidelines for constructing and improving DSLs. So called *fourth generation* programming languages – following third generation hardware-independent general-purpose languages – are usually DSLs that provide higher levels of abstraction for a specific domain, such as data management, analysis and manipulation [Fowler 2010; Klepper and Bock 1995].

Heterogeneous computing In heterogeneous computing, distributed systems consist of different kinds of processing devices, supporting different specialized processing features. The OpenCL standard [Khronos OpenCL Working Group 2009] for implementing systems across heterogeneous platforms is rather low-level, requiring the programmer to be aware of the specific hardware, e.g., specifically redesigning serial algorithms into parallel ones. Approaches for improving programming heterogeneous systems include (i) compiler directives to offload computations to specialized processing units, independent of specific hardware characteristics [Andión et al. 2016], (ii) domain-specific embeddings for general-purpose languages [Breitbart 2009; Lawlor 2011; Viñas et al. 2013] abstracting over low level details, such as compute kernel execution, and (iii) higher level programming models that provide primitives for a predefined set of operations [Wienke et al. 2012].

Big data processing systems Part of the success of modern Big Data systems is due to a programming interface that – similar to multitier programming – allows developers to define components that run on different hosts in the same compilation

unit, with the framework adding communication and scheduling. This class of systems includes batch processing frameworks like Hadoop [Dean and Ghemawat 2008] and Spark [Zaharia et al. 2012], as well as stream processing systems like Flink [Alexandrov et al. 2014] and Storm [Apache Software Foundation 2011b]. Since queries may process datasets that span multiple data centers and minimizing the traffic is crucial, approaches like Silos [Kloudas et al. 2015] offer abstractions that group nodes belonging to the same location so that the scheduler can minimize cross-data-center data transfer. Yet, in Big Data systems, the language semantics is visibly different, e.g., mutable shared variables are transformed in non-shared separated copies.

Operator placement In contrast to explicit placement (e.g., via annotations), the operator placement problem is about finding the *best* host on which each operator should be deployed in a distributed system in order to maximize a certain metric, such as throughput [Cugola and Margara 2013; Lakshmanan et al. 2008] or load [Cherniack et al. 2003]. Methods in this field include the creation of overlay networks where operators are assigned to hosts via random selection [Huebsch et al. 2003], network modeling [Pietzuch et al. 2006] and linear optimization to find the optimal solution to the constraint problem [Cardellini et al. 2016]. These systems adopt operators as the deployment unit. To reduce the load of the placement algorithm, Zhou et al. [2006] propose a coarser granularity and deploy *query fragments*, i.e., groups of operators.

3 A Multitier Language Design: ScalaLoci

> *The language seems carefully chosen to avoid any loopholes.*
>
> **— Data**

This chapter introduces the ScalaLoci programming language for developing distributed systems featuring *placement types*, a novel language abstraction for assigning locations to data and to computations, which enables the design of distributed applications in their entirety and thus allows reasoning about the complete system. ScalaLoci provides in-language specification of distributed architectures, and type-safe data flow specification over multiple hosts. After laying out the design space, we present the core abstractions of ScalaLoci and demonstrate ScalaLoci through examples. Finally, we describe ScalaLoci's fault tolerance mechanism and its execution model.

3.1 Design Considerations

ScalaLoci's design unfolds around three major ideas. First, the distributed topology (i.e., separate locations that execute code) is explicit and specified by the programmer who assigns code to each system component. Second, remote communication within the distributed system (i.e., with performance and failure characteristics different from local communication) is explicit and supports event-based interaction between components. Third, ScalaLoci abstractions are embedded as domain-specific features into an existing general purpose language. We lay out the design space for our language implementation.

Explicit specification of distribution and topology Some languages for developing distributed systems abstract over the topology, i.e., code is agnostic to the system's components and their connections. These languages cannot exploit developer knowledge about the system topology. Such approach is often chosen for highly specialized programming models, e.g., streaming systems [Dean and Ghemawat 2008; Zaharia et al. 2012], intentionally hiding distribution. On the other

end lie approaches where developers specify the topology. Distribution unavoidably becomes apparent when implementing different components, e.g., in actor systems [Agha 1986]. While actor systems encapsulate components into actors, topological information is not encoded at the language level but managed explicitly by the developer.

Clearly, a multitier programming model for developing generic distributed systems – similar to what the actor model offers – cannot abstract over distribution completely. To decide on the component that executes a specific part of the code, the compiler or the runtime may determine the location automatically by splitting code statically or dynamically, i.e., the developer gives up control over where code is executed. Alternatively, developers can annotate parts of the code with the location where it should be executed.

We decided for an annotation-based approach that gives the developer full control over the distribution, making locations explicit in the language. We think that it is essential for the developer to restrict the locations where code is executed for comprehending the system's behavior, e.g., regarding performance or privacy. Mixing automatic and explicit placement of values remains to be explored further.

Encoding the topology statically allows for providing static guarantees, such as that remote communication adheres to the topology and is statically type-checked. Thus, we (i) make location annotations part of a value's type and (ii) encode the topology specification at the type level, which allows the compiler to check correctness of remote accesses and ensures type safety across distributed components.

Explicit remote access and event-based communication Communication in distributed systems often follows a message passing approach (such as in actor systems). Message passing mechanisms, however, are quite low level with explicit send and receive operations [Gorlatch 2004], which disrupt control flow between a sending and a receiving side. Further, message loops often need to pattern-match on received messages since messages are essentially untyped. Instead, we favor strongly typed communication mechanisms.

Message passing semantics is close to sending packets over the network whereas communication mechanisms like remote procedure calls are closer to abstractions commonly found in high-level languages. For remote procedures, however, it is important to consider the inherent differences between remote calls and local calls [Kendall et al. 1994]. In contrast to local calls, remote calls need to account for network latency and potential communication failures. For this reason, in our design, remote calls return a value that represents an asynchronous computation which may potentially fail (e.g., a future).

Further, distributed applications are often event-based [Carzaniga et al. 2001; Meier and Cahill 2002]. For example, web clients send events to the server to trigger computations or cause a change to persistent storage. Servers send events to reflect such changes on the clients. Hence, we support specifying data flow in a declarative way using reactive programming abstractions.

Embedding There is a trade-off between designing a new language from scratch, which provides the greatest amount of flexibility, and embedding new abstractions into an existing language, which restricts the design space since the host language dictates the underlying feature set and available syntax. On the other hand, embeddings enable leveraging existing libraries and tooling.

We decided to embed our abstractions into Scala for its powerful type system, which can encode the topology specification and the placement of values. Our approach retains compatibility with plain Scala and preserves access to the Scala (and Scala.js) ecosystem. We exclusively use syntactically valid and type-correct Scala, allowing for expressing the placement of values in their types. The special semantics of placed values is implemented by a compile-time transformation based on macro expansion.

3.2 Programming Abstractions of ScalaLoci

In this section, we introduce the programming abstractions provided by ScalaLoci. The next section demonstrates how their combination significantly simplifies developing distributed systems.

3.2.1 In-Language Architecture Definition

In ScalaLoci, the architectural scheme of a distributed system is expressed using *peers* and *ties*. Peers and are encoded as abstract types and represent the different kinds of components of the system. Ties specify the kind of relation among peers. Remote access is only possible between tied peers. For instance, the client–server architecture is defined by a *server* peer and a *client* peer:

```
1 @peer type Client <: { type Tie <: Single[Server] }
2 @peer type Server <: { type Tie <: Single[Client] }
```

Both peers have a *single tie* to each other, i.e., clients are always connected to a single server instance and each corresponding server instance always handles a single client. To allow the definition of different architectural schemes, a multiplicity is associated with each tie: *multiple* is the most general multiplicity for a tie, *optional*

and *single* are specializations. A variant of the client–server model, where a single server instance handles multiple clients, is modeled by a *single tie* from client to server and a *multiple tie* from server to client:

```
@peer type Client <: { type Tie <: Single[Server] }
@peer type Server <: { type Tie <: Multiple[Client] }
```

We introduce ScalaLoci's abstractions with a P2P chat example, where nodes connect directly among themselves and every node maintains a one-to-one chat with every other connected remote node. In a P2P architecture, every peer instance can be represented by the same `Node` peer – in P2P, peers are homogeneous. The `Node` peer has a *multiple tie* to itself since nodes in a P2P system can maintain connections to an arbitrary number of other nodes. A `Registry` peer is used to discover other nodes. After discovery, nodes do not need to remain connected to the registry. Hence, their relation to the registry is an *optional tie*:

```
@peer type Registry <: { type Tie <: Multiple[Node] }
@peer type Node <: { type Tie <: Multiple[Node] with Optional[Registry] }
```

In case a node has more than one tie, ties are expressed by a compound type, e.g., `Multiple[Node] with Optional[Registry]` is a *multiple tie* to a `Node` peer and an *optional tie* to a `Registry` peer. Thus, peers and ties can specify complex schemes. *Multiple* ties are the most general case. *Optional* ties model a channel that may not be open for the complete up time of the application, forcing the developer to explicitly deal with such case in the application code. For instance, in our P2P chat example, nodes do not need to stay connected to the registry for chatting with other nodes, hence their tie is optional. Restricting the tie to *single* removes the need of handling the case that no remote peer instance is connected.

A peer abstracts over instances of the same kind. Yet, during execution, multiple peer instances, e.g., multiple nodes of peer type `Node`, can inhabit the system and dynamically connect to other peer instances at run time. ScalaLoci allows distinguishing among them using *remote peer references*. Peer instances of the same type can be accessed uniformly via *value aggregation* (Section 3.2.5 on page 59).

3.2.2 Placement Types

Based on the peers defined in the system architecture, ScalaLoci allows specifying *placed* data and computations. Placement is statically defined and part of a value's type. Placed values of type `T on P` represent a value of type `T` placed on a peer `P`, where `T on P` is infix notation for the parameterized type `on[T, P]`. In our P2P chat application, for example, each `Node` peer defines an event stream of outgoing messages, i.e., the stream is *placed* on the `Node` peer:

```
val messageSent: Event[String] on Node = /* stream of outgoing messages */
```

Instead of explicitly annotating a value with a placement type, it can be initialized using the `on[P] { e }` notation, inferring the placement type `T on P`. While the type of a placed value `T` can be inferred, its placement `P` is always explicit since we want developers to consciously decide on placement:

```
1 val messageSent = on[Node] { /* stream of outgoing messages */ }
```

Event streams of type `Event[T]` and time-changing values of type `Signal[T]` are part of ScalaLoci's reactive abstractions (Section 3.2.4 on the next page). The `messageSent` event stream is accessible remotely from other peers to receive the chat messages. Remote visibility of placed values can be regulated: Placed values denoted by the type `T on P` specify that values placed on a peer `P` can be accessed from other peers. Placed values of type `Local[T] on P` specify values that can only be accessed locally from the same peer instance, i.e., they cannot be accessed remotely over the network. In the P2P chat, the `Registry` peer maintains an index of all participants. The index is defined local to the `Registry` peer to prevent participants from directly accessing the index:

```
1 val participantIndex: Local[Index] on Registry = /* users registry */
```

3.2.3 Remote Access

Accessing remote values (values on another peer instance) requires the `asLocal` syntactic marker. With `asLocal`, we remind developers that a remote access creates a *local* representation of the value (e.g., by transmitting it over the network or by establishing a remote dependency) that can then be used locally. There are two reasons for making remote communication explicit: First, we want to raise the developers' awareness of whether they are accessing a local or a remote value since (i) local and remote accesses have completely different performance characteristics regarding latency and (ii) remote invocation can potentially fail due to network communication failures. Second, having explicit syntax for remote access allows selecting different aggregation schemes, i.e., different ways of abstracting over multiple remote peer instances. For example, we use `asLocal` to access the remote value of a single remote peer instance and `asLocalFromAll` to access the remote values of multiple remote peer instances in case of a multiple tie (Section 3.2.5 on page 59).

The following example demonstrates the remote access to the `messageSent` event streams of the chat partners. We assume the existence of a `joinMessages` function,

which collects all incoming messages into a growing list representing the log of received messages:

```
1 val receivedMessages: Signal[List[String]] on Node =
2   joinMessages(messageSent.asLocalFromAll)
```

Thus, this code snippet makes the messages sent from remote peer instances (`messageSent`) locally available (`asLocalFromAll`) as the list of received messages (`receivedMessages`). Since `receivedMessages` is placed on `Node`, the expression `joinMessages(messageSent.asLocalFromAll)` is evaluated on each `Node` instance. As for the previous declaration, `messageSent` is also placed on `Node`. Thus, every `Node` instance provides a `messageSent` event stream. Since the architecture defines a multiple `Node`-to-`Node` tie, the remote access from a `Node` instance to another `Node` instance using `asLocalFromAll` is correct. In ScalaLoci, peers, ties, and placements are known statically and the compiler checks that access to remote values is consistent with the architecture definition. For instance, the following code shows an invalid remote access, which is statically rejected by the compiler:

```
1 val remoteIndex: Index on Registry =
2   participantIndex.asLocal
```

The remote access via `asLocal` from the `Registry` peer to the `participantIndex` value, which itself is placed on `Registry`, does not type-check for two reasons: (i) the `participantIndex` is not accessible remotely since it is a local placed value and (ii) the remote access violates the architecture specification, which does not specify a `Registry`-to-`Registry` tie. Although ScalaLoci favors more high-level communication mechanisms, we also support traditional remote procedure calls using the `remote call` *`proc`*`(`*`args`*`)` construct, calling the *`proc`* method with the given arguments *`args`* remotely.

The semantics of accessing remote values depends on their type. By default, accessing primitive values and collections remotely features copy semantics – changes are not reflected remotely. Accessing reactive remote values establishes a remote *depends-on* relation (Section 3.2.4). The semantics is in fact open and defined by *transmitters* (Section 6.5.3 on page 122).

3.2.4 Multitier Reactives

In the running example of our P2P chat, we already defined the `messageSent` *event* and the `receivedMessages` *signal*. Events and signals are ScalaLoci's *reactive abstractions* (a.k.a. *reactives*) in the style of systems like REScala [Salvaneschi et al. 2014b]. Events model discrete changes. The following code defines an event stream

of type `Event[String]`, which is a sequence of events carrying a `String` each, and pushes a new event into the stream:

```
val messageSent: Event[String] = Event[String]()
messageSent.fire("some message")
```

Signals model continuous time-changing values which are automatically updated by the language runtime. A signal `val s3: Signal[Int] = Signal { s1() + s2() }`, for instance, depends on input signals `s1` and `s2`. The *signal expression* `s1() + s2()` is recomputed to update `s3` every time `s1` or `s2` changes. Signals inside `Signal {...}` expressions require `()` to access their current value and register them as dependencies. Events and signals can interoperate. The following snippet defines a signal that holds a list of all sent messages:

```
val sentMessages: Signal[List[String]] =
  messageSent.fold(List.empty[String]) { (log, message) => message :: log }
```

Event streams support operations such as folding, mapping, filtering and handler registration [Maier and Odersky 2013]. For example, the `list` operator can be used to define the signal `sentMessages` as `messageSent.list` instead of using the `fold` operator to fold over the event stream as shown in the code snippet above. Defining reactives based on other reactives establishes a *depends-on* relation between them, e.g., `sentMessages` depends on `messageSent`, allowing the definition of complex data flow graphs with transitive dependencies.

ScalaLoci embraces asynchronous remote communication, which is the default in many distributed systems for performance and decoupling. Remote access to a reactive via `asLocal` creates a local representation of the reactive and extends the data flow graph without blocking. Asynchronicity only becomes visible when imperative code interfaces with the reactive system, e.g., a method invocation that fires an event may return before the event reaches remote dependents. Instead, in code written in reactive programming style, propagating values asynchronously is transparent to the user.

Our consistency model corresponds to the one commonly used in actor systems and implemented by Akka [Lightbend 2009a]: update propagation to remote reactives feature at-most-once delivery and order preservation for sender–receiver pairs (cf. Section 3.5 on page 70). Local propagation is *glitch-free* [Cooper and Krishnamurthi 2006].

3.2.5 Peer Instances

Architecture definitions specify peer *types* and their relation, but the number of connected peer *instances* can change over time. A specific instance of type `P` is

identified by a remote peer reference of type `Remote[P]`, which also offers information about the state of the network connection and underlying network protocol – including protocol-specific properties, e.g., host and port for TCP.

The `remote[P].connected` signal provides a time-changing list of currently connected peer instances of type `P`. User code is informed about established or lost connections via the `remote[P].joined` and `remote[P].left` events. It is possible to setup connections programmatically at run time using `remote[P].connect` and providing the address at which the remote peer instance is reachable.

Since peers abstract over multiple instances of the same peer type, remote access to a value `T on P` may refer to the values of multiple instances of type `P`. When accessing placed values remotely, ScalaLoci offers two options to handle such multiplicity: (i) use an aggregated value over all remote peer instances (Section 3.2.6) or (ii) access a single value of the desired peer instance (Section 3.2.7 on the next page).

3.2.6 Aggregation

Accessing a remote value using a variant of `asLocal` reads a value from each connected peer instance. The actual version of `asLocal` that is invoked depends on (i) the type of the value, (ii) the placement of the value, (iii) the placement of the expression accessing the value and (iv) the tie that connects the respective peers. For a single tile, reading a remote value just results in a local representation of the value. In case of a multiple tie, reading a remote value abstracts over connected remote peer instances and aggregates their value.

The aggregation scheme depends on the type of the value. Accessing remote primitives and standard collections from a remote peer instance with a multiple tie returns a value of type `Seq[(Remote[P], Future[T])]`. This value provides a future [Baker and Hewitt 1977] – which is part of Scala's standard library – for each remote peer instance. The future values account for network latency and possible communication failures by representing a value which may not be available immediately, but will become available in the future or produce an error.

Accessing remote reactives (i.e., events or signals) from a remote peer instance with a multiple tie creates a dependency to each accessed reactive. Upon remote access, changes of the remote reactives are propagated to the accessing peer. Accessing a remote reactive yields a value of type `Signal[Seq[(Remote[P], Signal[T])]]` for signals and of type `Signal[Seq[(Remote[P], Event[T])]]` for events. The outer signal changes when the list of connected peers does. The inner reactive corresponds to the reactive on each peer instance `Remote[P]`. Reactives subsume the functionalities of futures, i.e., the propagation of remote values is asynchronous

(cf. Section 3.2.4 on page 58) and reactives can propagate failures (cf. Section 3.4 on page 68).

To make aggregation explicit, we require an `asLocalFromAll` variant to access multiple remote instances uniformly. The following code snippet accesses the `messageSent` event stream of our P2P chat to aggregate all incoming messages from all the chat partners:

```
1 val incoming: Signal[Seq[(Remote[Node], Event[String])]] on Node =
2   messageSent.asLocalFromAll
```

We additionally provide `asLocalFromAllSeq` for placed event streams, which aggregates the event occurrences of all connected remote peer instances into a single local event stream, providing all event occurrences sequentially in the single stream. This access pattern is often more convenient for event streams, which represent discrete occurrences:

```
1 val incomingSeq: Event[(Remote[Node], String)] on Node =
2   messageSent.asLocalFromAllSeq
```

In Section 3.3 on page 63, we demonstrate how aggregation simplifies treatment of several remote instances of equal type (Listing 3.6 on page 67).

3.2.7 Subjective Access

Remote peer references `Remote[R]` distinguish among peer instances at run time. Complementing aggregation, the `from` operator is used to access a remote value from a specific peer instance only. The following example shows the remote access to the `messageSent` stream of the single peer instance given by `node`. The messages are then stored in a time-changing list of type `Signal[List[String]]` using the `list` operator on event streams:

```
1 def messageLog(node: Remote[Node]): Local[Signal[List[String]]] on Node =
2   (messageSent from node).asLocal.list
```

Sometimes, selecting data *on the receiver*, like `from` does, is inefficient or insecure. Hence, dually to `from`, ScalaLoci provides declarative sender-side selection via *subjective* values – denoted by the subjective placement type `T per P' on P`. By default, every peer instance accessing a remote value reads the same content. In contrast, subjective values exhibit a different value based on the accessing peer instance. The definition of a subjective value binds an identifier holding a reference to the peer instance that accesses the subjective value. Using this identifier, user code can filter the value on a per remote peer instance basis before the value leaves the local instance. In the P2P chat, every participant can take part in multiple chats simultaneously, but messages typed by a user should be sent only to the currently

selected partner. We achieve this goal by declaring the `messageSent` event stream of outgoing messages *subjective*:

```
1 val messageSent: Event[String] per Node on Node = node: Remote[Node] =>
2   ui.messageTyped filter { msg => ui.isSelectedChat(node) }
```

The `node` identifier is used to filter the `ui.messageTyped` event stream based on the accessing peer instance for defining the `messageSent` stream that only contains the messages for the `node` chat partner. Crucially, when accessing a subjective declaration, each peer instance "sees" a different `messageSent` event stream containing only the messages directed to it. Alternatively, subjective values can be defined using the `sbj` modifier on the `on[P]` notation, inferring the subjective placement type `T per P' on P`:

```
1 val messageSent = on[Node] sbj { node: Remote[Node] =>
2   ui.messageTyped filter { msg => ui.isSelectedChat(node) }
3 }
```

Symmetrically, a remote peer reference can be used to retrieve the local value which is (subjectively) selected for a specific peer. For example, we define a reactive list `sentMessages` containing messages sent to the remote peer `node` with `messageSent to node`. The call to `list` creates a signal of a list which grows with each event occurrence:

```
1 def sentMessages(node: Remote[Node]): Local[Signal[List[String]]] on Node =
2   (messageSent to node).asLocal.list
```

3.2.8 Remote Blocks

Distributed systems often require offloading data processing to other hosts. To this end, ScalaLoci provides *remote blocks* – expressions executed on remote peer instances. The value from the remote evaluation of the block is returned to the instance executing the block. A remote block expression `on[P].run { e }` runs the computation `e` on every connected peer instance of type `P` and and `on(p).run { e }` runs the computation on the given instance `p`. Remote blocks only capture (close around) values stated explicitly via `capture` and – like remote access via `asLocal` (Section 3.2.3 on page 57) – feature copy semantics for non-reactive values. Implicit captures, which may be unintentional, are compilation errors.

Listing 3.1 on the next page shows an excerpt from our P2P chat example to setup the connection between two peer instances dynamically at run time, which have no direct connection yet. The initiating node transfers the connection request to the requested node through the central registry server, which maintains a connection to both nodes. The initiating node passes the value of `requestedId` to the registry via a remote block (Line 4). The remote block is dispatched *subjectively* (Section 3.2.7

Listing 3.1 Remote blocks.

```
1  on[Node] {
2    val requestedId = /* ... */
3
4    on[Registry].run.capture(requestedId) sbj { requesting: Remote[Node] =>
5      val requested = participantIndex.getRemote(requestedId)
6      val address = participantIndex.getAddress(requesting)
7
8      on(requested).run.capture(address) {
9        remote[Node].connect(address)
10     }
11   }
12 }
```

on page 61), i.e., the `requesting` identifier (Line 4) is bound to a peer remote reference to the initiating node. The registry resolves the remote reference for the requested peer and the address at which the node is reachable from an internal `participantIndex` (Lines 5 and 6). The registry then runs a remote block on the requested peer instance (Line 8), which connects to the initiating node (Line 9) by setting up a dynamic connection via `remote[P].connect` (cf. Section 3.2.5). From this time on, the peers can communicate directly without a central server.

Remote blocks do not pose a security threat as their semantics does not entail mobile code. They delimit a functionality executed on other peer instances in a concise way. No code is sent over the network, only the values exchanged between the block and the surrounding environment, i.e., the captured values and the result value. The code that a remote block executes remotely is statically known. For example, for `on[PeerA] { val v = x; on[PeerB].run.capture(v) { val y = v } }`, the code separation into peer-specific code places `val v = x` on `PeerA` and `val y = v` on `PeerB`. This placement is fixed after compilation. Only the value `v` – which is explicitly captured – is sent from the instance of `PeerA` to an instance of `PeerB`.

3.3 ScalaLoci in Action

In this section, we demonstrate with several *complete* applications how the abstractions introduced in Section 3.2 on page 55 help defeating the complexity of developing distributed systems. By supporting multitier reactivity at the language level, data flows among over multiple peers can be declaratively specified.

Multitier code is syntactically valid and type-correct Scala code. The `@multitier` annotation is used to mark classes, traits or objects as multitier modules (cf. Section 4.1.1 on page 76) and gives placement types special semantics through peer-based splitting (Section 6.4.2 on page 112).

Listing 3.2 Messaging logic for multiple P2P chats.

```
1  @multitier object P2PChat {
2    @peer type Node <: { type Tie <: Multiple[Node] }
3
4    type SingleChatLog = Signal[List[String]]
5    type MultiChatLogs = Signal[List[SingleChatLog]]
6
7    val ui: UI on Node = UI()
8
9    val messageSent = on[Node] sbj { node: Remote[Node] =>
10     ui.messageTyped filter { _ => ui.isSelectedChat(node) }
11   }
12
13   def messageLog(node: Remote[Node]): Local[SingleChatLog] on Node =
14     ((messageSent from node).asLocal ||
15      (messageSent to node)).list
16
17   val chatLogs: MultiChatLogs on Node =
18     remote[Node].joined.fold(List.empty[SingleChatLog]) { (chats, node) =>
19       messageLog(node) :: chats
20     }
21 }
```

Chat application: Messaging Listing 3.2 concludes the P2P chat example showing the complete messaging logic. We only leave out the logic for dynamically setting up connections, which is already in Listing 3.1 on the previous page. The application logs messages that are sent and received by each participant as a composition of the data flow from the local UI and from remote chat partners. In the example, nodes are connected to multiple remote nodes and maintain a one-to-one chat with each. Users can select any chat to send messages.

The `messageSent` (Line 9) event is defined as subjective value (Section 3.2.7 on page 61) filtering the `ui.messageTyped` messages from the UI (Line 10) for the currently active chat partner `node`. The `messageLog` (Line 13) signal contains the chat log for the chat between the local peer instance and the remote `node` given as parameter. It merges the remote stream for the chat messages *from the remote instance* `node` (Line 14) and the local stream *subjective to the remote instance* `node` (Line 15) via the `||` operator. The event stream resulting from such merge fires whenever either of both operands fires. The chat log is a signal created using `list`, which extends the list by an element for each new event occurrence in the merged stream. The `chatLogs` signal folds (Line 18) the `remote[Node].joined` event stream (cf. Section 3.2.5 on page 59), which is fired for each newly connected chat partner, into a signal that contains the chat logs for every chat partner generated by calling `messageLog` (Line 19).

Listing 3.3 Tweet processing pipeline.

```
1  @multitier object TweetProcessing {
2    @peer type Input <: { type Tie <: Single[Filter] }
3    @peer type Filter <: { type Tie <: Single[Mapper] with Single[Input] }
4    @peer type Mapper <: { type Tie <: Single[Folder] with Single[Filter] }
5    @peer type Folder <: { type Tie <: Single[Mapper] }
6
7    val tweetStream: Event[Tweet] on Input =
8      retrieveTweetStream()
9
10   val filtered: Event[Tweet] on Filter =
11     tweetStream.asLocal filter { tweet => tweet.hasHashtag("multitier") }
12
13   val mapped: Event[Author] on Mapper =
14     filtered.asLocal map { tweet => tweet.author }
15
16   val folded: Signal[Map[Author, Int]] on Folder =
17     mapped.asLocal.fold(Map.empty[Author, Int].withDefaultValue(0)) {
18       (map, author) => map + (author -> (map(author) + 1))
19     }
20 }
```

Tweets Next, we show how the operators in a processing pipeline can be placed on different peers (Listing 3.3) to count the tweets that each author produces in a tweet stream. The application receives a stream of tweets on the `Input` peer (Line 8), selects those containing the `"multitier"` string on the `Filter` peer (Line 11), extracts the `author` for each tweet on the `Mapper` peer (Line 14) and stores a signal with a map counting the tweets from each author on the `Folder` peer (Line 18).

Email application Listing 3.4 on the next page shows a client–server e-mail application. The server stores a list of e-mails. The client can request the e-mails received in the n previous days containing a given word. The client user interface displays the e-mails broken into several pages. If the word is not in the current page, the user is informed.

The definition of the `word` signal of type `Signal[String] on Client` (Line 5) defines a signal carrying strings placed on the `Client` peer. Thanks to multitier reactives, the client-side signal `inCurrentPage` (Line 20) is defined by the composition of the local client-side signal `word` and the remote server-side signal `filteredEmails`. The latter (Line 11) is defined as a composition of a local signal (Line 13) and two remote signals (Lines 14 and 15).

Token ring We model a token ring (Listing 3.5 on page 67), where every node in the ring can send a token for another node. Multiple tokens can circulate in the

Listing 3.4 Email application.

```
1  @multitier object MailApp {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4
5    val word: Signal[String] on Client = /* GUI input */
6
7    val days: Signal[Int] on Client = /* GUI input */
8
9    val allEmails: Local[Signal[List[Email]]] on Server = /* e-mail collection */
10
11   val filteredEmails: Signal[List[Email]] on Server =
12     Signal {
13       allEmails() filter { email =>
14         (email.date >= Date.today() - days.asLocal()) &&
15         (email.text contains word.asLocal())
16       }
17     }
18
19   val inCurrentPage: Local[Signal[Boolean]] on Client =
20     Signal { isCurrentFirstPage(word(), filteredEmails.asLocal()) }
21 }
```

ring simultaneously until they reach their destination. Every node has exactly one predecessor and one successor.

We define a `Prev` and a `Next` peer and specify that a `Node` itself is both a predecessor and a successor and has a single tie to its own predecessor and a single tie to its successor. Using *multiple* ties would allow nodes to join and leave, updating the ring dynamically but is not discussed further. Tokens are passed from predecessors to successors, hence nodes access the tokens sent from their predecessor. For this reason, values are placed on the `Prev` peer. Every node has a unique ID (Line 8). The `sendToken` event (Line 10) sends a token along the ring to another peer instance. The `recv` event stream (Line 13) provides the data received by each peer instance. Each node fires `recv` when it receives a token addressed to itself, i.e., when the `receiver` equals the node ID (Line 14), and forwards other tokens. The expression `sent.asLocal \ recv` (Line 16) evaluates to an event stream of all events from `sent.asLocal` for which `recv` does not fire. Merging such stream (of forwarded tokens) with the `sendToken` stream via the `||` operator injects both new and forwarded tokens into the ring.

Master–worker We now show a ScalaLoci implementation of the master–worker pattern (Listing 3.6 on the next page) where a master node dispatches tasks – double a number, for simplicity – to workers. The `taskStream` on the master (Line 8) carries the tasks (Line 5) as events. The `assocs` signal (Line 10) contains the assignments of workers to tasks. It folds over the `taskStream || taskResult.asLocalFromAllSeq`

Listing 3.5 Token ring.

```
@multitier object TokenRing {
  @peer type Prev <: { type Tie <: Single[Prev] }
  @peer type Next <: { type Tie <: Single[Next] }
  @peer type Node <: Prev with Next {
    type Tie <: Single[Prev] with Single[Next]
  }

  val id: Id on Prev = Id()

  val sendToken: Local[Event[(Id, Token)]] on Prev =
    Event[(Id, Token)]()

  val recv: Local[Event[Token]] on Prev =
    sent.asLocal collect { case (receiver, token) if receiver == id => token }

  val sent: Event[(Id, Token)] on Prev = (sent.asLocal \ recv) || sendToken
}
```

Listing 3.6 Master–worker.

```
@multitier object MasterWorker {
  @peer type Master <: { type Tie <: Multiple[Worker] }
  @peer type Worker <: { type Tie <: Single[Master] }

  case class Task(v: Int) { def exec(): Int = 2 * v }

  // to add tasks: `taskStream.fire(Task(42))`
  val taskStream: Local[Event[Task]] on Master = Event[Task]()

  val assocs: Local[Signal[Map[Remote[Worker], Task]]] on Master =
    (taskStream || taskResult.asLocalFromAllSeq).fold(
        Map.empty[Remote[Worker], Task],
        List.empty[Task]) { (taskAssocs, taskQueue, taskChanged) =>
      assignTasks(taskAssocs, taskQueue, taskChanged, remote[Worker].connected)
    }

  val deployedTask: Signal[Task] per Worker on Master =
    worker: Remote[Worker] => Signal { assocs().get(worker) }

  val taskResult: Event[Int] on Worker =
    deployedTask.asLocal.changed collect { case Some(task) => task.exec() }

  val result: Signal[Int] on Master =
    taskResult.asLocalFromAllSeq.fold(0) { case (acc, (worker, result)) =>
      acc + result
    }
}
```

event stream that fires for every new task (`taskStream`) and every completed task (`taskResult.asLocalFromAllSeq`). We assume the existence of an `assignTasks` method (Line 14), which assigns a worker to the new task (`taskAssocs`) or enqueues the task if no worker is free (`taskQueue`) based on the folded event (`taskChanged`) and the currently connected worker instances (`remote[Worker].connected`). The `deployedTask` signal (Line 17) subjectively provides every worker instance with the task it is assigned. Workers provide the result in the `taskResult` event stream (Line 20), which the master aggregates into the `result` (Line 23) signal. The signal is updated for every event to contain the sum of all values carried by the events.

3.4 Fault Tolerance

In distributed systems, components can fail anytime. To handle failures, we provide a mechanism that unifies reactives and supervision à la actors. The key idea is that the *depends-on* relation between reactives establishes a *supervisor-of* relation, where the supervisor is notified if a supervised reactive fails. For example, in `e.filter(_ > 10).map(_.toString)`, the `map` reactive depends on the `filter` reactive and `map` supervises `filter`.

Error propagation Signals and events can carry the successfully computed value val or an error value err upon failure. If, during reevaluation, a reactive accesses an err value of a reactive it depends on, the recomputation *fails* – i.e., the recomputation is skipped, like in Akka actors [Lightbend 2009e] – and the reactive also emits err. Propagation of errors along the data flow graph is in line with the approaches taken by REScala [Mogk et al. 2018a], Rx [Meijer 2010] and Reactive Streams [Reactive Streams Initiative 2014] – and the Reactive Streams implementations RxJava and Akka Streams. Signals, which hold a value continuously, can handle computation failures by replacing the err value with a value val (`s.recover { error => val }`). Events can additionally ignore failures (`e.recover { error => None }`) or also handle them by replacing the err value (`e.recover { error => Some(val) }`). A reactive carrying err holds a success value val again as soon as it reevaluates to val upon new input. Thus, a failed computation does not prevent processing further input.

Supervision To supervise a reactive computation *r*, a developer can simply declare another reactive that depends on *r*, which then becomes the supervisor, like in actors, and receives an err value if the supervised reactive fails. A supervisor can (i) ignore notifications of failure or (ii) handle them implementing a recovery strategy.

Our mechanism supports the most common cases for reactives, ScalaLoci's main communication abstraction, still retaining the full generality of supervision relations

that proved effective in the actor model. This mechanism allows monitoring reactive computations that are not necessarily arranged as trees – trees are a special case. Similarly, Akka supports monitoring schemes beside supervision trees that allow arbitrary monitoring relations [Lightbend 2009d]. For streams, for example, an application can neglect the cases that produce a failure – e.g., with spurious data in big data analytics – generate default values that track failed cases, or check the successful processing of an event through a complete stream pipeline by creating a stream of acknowledgments (or err values) from the sink to the source (cf. Section 3.5 and Listing 3.8 on the following page and on page 72). Generality is achieved building on top of the err value propagation/supervision mechanism to allow custom fault handling strategies. For example, a supervisor can emit an event inducing all reactives in the system to reset their internal state, the equivalent of the `one_for_all` Erlang recovery strategy [Ericsson 1987b], effectively restarting the reactive system to handle a failure. Failures in the generated communication layer occur in case a remote connection breaks or cannot be established. Accessing a remote reactive which is not connected (anymore) also propagates an `err` value, thus making user code aware of communication failures.

Fault-tolerant master–worker We demonstrate our approach augmenting the master–worker example (Listing 3.6 on page 67) with fault handling. A first improvement is that the master simply ignores tasks that cause a worker to fail. This is achieved by dropping err values before they propagate to `fold` in Line 24. The `result` signal depends on the workers' `taskResult` event stream through `taskResult.asLocalFromAllSeq`, which performs event aggregation (Section 3.2.4 on page 58) *and* establishes a supervision relation. A small change to the `result` signal (in gray) suffices:

```
1 taskResult.asLocalFromAllSeq.recover{ error => None }.fold(...)
```

After merging events from all workers into a single event stream, all err values are dropped from the stream and `fold` processes only successfully computed values. A second improvement is to introduce a stream of failed tasks for diagnostics:

```
1 val failedTasks: Report on Master =
2   taskResult.asLocalFromAllSeq
3     .recover { error => Some(Report(assocs().get(error.remote))) }
4     .collect { case report @ Report(_) => report }
```

The `failedTasks` stream first replaces events carrying an `err` value by a `Report` value using the `recover` operator. The report contains the task associated with the disconnected worker remote peer instance (via `assocs`). Finally, reports are collected, filtering out successfully completed tasks.

Machine failures or network connection losses affect all reactives on the lost peer instance. To react to a disconnection of a remote peer instance, peer instances monitor the peer instances to which they are connected. A peer instance is informed about a disconnection via the `remote[P].left` event (Section 3.2.5 on page 59) and can take countermeasures. This mechanism is similar to how Akka detects that the communication to a remote actor fails: the monitoring actor receives a `Terminated` message for the monitored actor [Lightbend 2009c].

In a third improvement, the master reassigns the task that was running on the disconnected worker to another worker via an event stream of tasks to redeploy:

```
1 val redeploy = Event[Task] on Master =
2   remote[Worker].left map { worker => assocs().get(worker) }
```

The `redeploy` event stream maps every disconnected worker remote reference (provided by `remote[Worker].left`) to the task to which the respective worker was assigned (via `assocs`), resulting in a stream of tasks that were assigned to disconnected worker instances. The `assocs` signal (Line 10 of Listing 3.6 on page 67), computing the assignments of workers to tasks, also needs to be updated to consider the tasks to be redeployed, additionally folding over the `redeploy` event stream (Line 11):

```
1 (taskStream || taskResult.asLocalFromAllSeq || redeploy).fold(...)
```

3.5 Execution Model and Life Cycle

Peer Startup In the previous sections, we have shown several multitier applications that define the distributed components inside a single module. We use the `@multitier` annotation on Scala objects to *instantiate* such multitier modules. To start up a distributed system, however, we also need to *start peers* defined in the modules. Different peer instances are typically started on different hosts and connect to each other over a network according to the architecture specification. As a consequence, an additional step is required to start the peers of (already instantiated) modules.

Listing 3.7 on the facing page shows the setup for a client–server application. We follow the idiomatic way of defining an executable Scala application, where an `object` extends `App` (Lines 6 and 11). The object body is executed when the application starts. The code executed when staring a Scala application is standard (non-multitier) Scala, which, in our example, uses `multitier start new Instance[...]` to start a peer of an instantiated multitier module. Line 1 instantiates a `MyApp` module. The `Server` *listens* on the tie towards the `Client` using TCP and port 1099

Listing 3.7 Peer startup.

```
1  @multitier object MyApp {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4  }
5
6  object ServerApp extends App {
7    multitier start new Instance[MyApp.Server](
8      listen[MyApp.Client] { TCP(1099) })
9  }
10
11 object ClientApp extends App {
12   multitier start new Instance[MyApp.Client](
13     connect[MyApp.Server] { TCP("example.com", 1099) })
14 }
```

(Line 8). The `Client` *connects* via the tie towards the `Server` using TCP and the same port and specifying the server address (Line 13).

Section 6.5.1 on page 120 elaborates on the supported underlying network protocols using *communicators*. For each tie, a peer instance can either initiate the communication (*connect*) or wait for incoming connections (*listen*). A peer instance can *connect* to a single remote instance on a single or optional tie and to an arbitrary number of remote instances on a multiple tie. In case a peer instance listens on a single or optional tie, a new local peer instance is created for each incoming connection. This is commonly the case in the client–server setting: a new server instance starts upon each request. In contrast, when a peer listens on a multiple tie, a single local peer instance handles all incoming connections. This approach supports architectures, e.g., (i) a web application, where each client is handled separately by the server, (ii) a server which handles client instances collectively or (iii) nodes connecting directly to each other in a P2P fashion. The dynamic counterpart to the automatic connection setup for establishing connections at run time is introduced in Section 3.2.5 on page 59, e.g., using `remote[Server].connect(TCP("example.com", 1099))`.

Consistency ScalaLoci adopts the consistency model found in many actor systems: (1) *at-most-once* delivery (i.e., delivery is not guaranteed and dropped messages are lost) for remote reactives and (2) order preservation for sender–receiver pairs [Ericsson 1987a; Lightbend 2009b]. Similar to actors, a developer can implement stronger consistency manually on top of ScalaLoci. For example, to achieve *at-least-once* delivery, events can be re-sent until they are acknowledged. In Listing 3.8 on the next page, a `send` event on `PeerA` transfers some payload of type `Data` to `PeerB`. A `timeout` event is regularly fired to implement the timeout for pending acknowledgments. The `msg` event (Line 10), which is a `send` event or a `resend` event after

Listing 3.8 At-least-once delivery on top of ScalaLoci.

```
1  val missing: Signal[Set[Data]] on PeerA =
2    (send || ack.asLocal).fold(Set.empty[Data]) {
3      case (missingAcks, Ack(v: Data)) => missingAcks - v
4      case (missingAcks, v: Data) => missingAcks + v
5    }
6
7  val resend: Event[Data] on PeerA =
8    timeout collect { case _ if missing().nonEmpty => missing().head }
9
10 val msg: Event[Data] on PeerA = send || resend
11
12 val ack: Event[Ack[Data]] on PeerB = msg.asLocal map { v => Ack(v) }
```

the timeout expired, is read on `PeerB` (Line 12). In the example, the `msg` event is sent back wrapped in an `Ack` acknowledgment (Line 12). On `PeerA`, we fold over the `send || ack.asLocal` event stream (Line 2), adding all sent events to the set of events awaiting acknowledgment (Line 4) and removing all acknowledged events from the set (Line 3). Every time `timeout` fires, the `resend` event (Line 7) fires for one of the events that are not acknowledged yet. For simplicity, in the example, we do not preserve ordering. It is possible to implement an order-preserving mechanism trading off performance for a higher level of consistency.

Given the abstraction level of ScalaLoci (reactives, multitier programming), however, we expect that developers do not implement higher consistency levels themselves. Hence, ScalaLoci allows developers to choose among different reactive systems with different levels of consistency (cf. Section 6.5.3 on page 122) We currently support several backends, e.g., Rx [Meijer 2010] as well as one that provides stronger consistency guarantees [Drechsler et al. 2014].

Execution order and concurrency Placed values are initialized at bootstrap on each peer instance. The evaluation of placed expressions adheres to the standard Scala semantics: variable declarations (`val` or `var`) are evaluated in definition order, lazy values (`lazy val`) are evaluated upon first access and method definitions (`def`) are evaluated on each access. ScalaLoci, by default, uses a single thread for evaluating values and computing remote blocks initiated by remote instances. This behavior can be adapted, e.g., using a thread pool to improve concurrent access. Sequential remote accesses from the same remote instance are evaluated in the same order but there are no guarantees for the remote access by different instances.

Cycles ScalaLoci allows defining software that entails distributed cycles, such as in the Token Ring application (Listing 3.5 on page 67). In this example, a node in the ring receives a token via the `sent` event from its predecessor and (i) either emits

it again via its `sent` event to be processed by the successor node (forming a cycle of `sent` events along the nodes in the ring) or (ii) processes the token and removes it from cycling through the ring. The reactive model used by ScalaLoci allows sending events along peer instances that are arranged in a cycle since messages are sent from one instance to another asynchronously (similar to messages in actor systems). After sending an event to a remote instance, the local instance continues processing incoming events. In particular, incoming events may be events that were originally sent by the local instance and are reaching the local instance again through a cycle. With this model, events can cycle around in the system being passed on to the next node until they are consumed and removed from the cycle.

Deployment To deploy a ScalaLoci application on the JVM, we generate Java bytecode that can be packaged into JAR files as usual. For web deployment, we generate Java bytecode for the server and JavaScript code for the client. The client code connects to the server using HTML5 WebSockets. ScalaLoci can potentially work with any web server to answer the initial HTTP request from the browser and serve the generated JavaScript code. We implemented example applications using Akka HTTP [Lightbend 2016] and the Play Framework [Lightbend 2007] for web applications as HTTP servers (cf. Sections 7.2 and 7.3 on page 126 and on page 131).

ScalaLoci peers on different hosts can be updated independently as long as there are no changes to the signature of placed values accessed by remote instances. Changing the signature of placed values requires that all affected peers are updated to avoid incompatibilities.

4 A Multitier Module System

> *Cross-connecting that many units will be tricky.*
>
> **— Geordi LaForge**

In this chapter, we describe the ScalaLoci module system. We present the design of multitier modules on the one hand. On the other hand, we demonstrate a number of examples for multitier modules and their composition mechanisms. We first introduce (concrete) *multitier modules* and show how they can be composed into larger applications. Then we present modules with *abstract peer types*, which enable the definition of abstract modules that capture a fragment of a distributed system, can be composed with other abstract modules and can eventually be instantiated for a concrete distributed architecture We show their composition through *module references* – references to other multitier modules – as well as another composition mechanism, *multitier mixing*. Next we show how such composition mechanism enables defining *constrained multitier modules*. Finally, we return to abstract peer types and demonstrate how they enable *abstract architectures*.

4.1 Multitier Modules

As ScalaLoci uses placement types to specify placement, i.e., developers can define new peers – components of the distributed system – by defining a peer type, our approach leverages the modularity and composability properties of types defining peer types as abstract type members of traits. By allowing developers to (i) freely define different peer types as part of a multitier module definition, and (ii) specializing peer types to peers of other modules, ScalaLoci enables abstracting over the concrete placement of values. Thus, placement in ScalaLoci is *logical*, not *physical*, i.e., a developer can divide a system into abstract places, modeled by ScalaLoci peers, and *merge* such places into the peers which are to be started when running the distributed system. The module system allows separating modularization and distribution, which are traditionally linked together due to the lack of distinct language abstractions for reasoning about distribution.

By embedding the ScalaLoci module system into Scala, Scala traits represent modules – adopting Scala's design that unifies object and module systems [Odersky

and Zenger 2005]. Traits can contain abstract declarations and concrete definitions for both type and value members – thus serve as both module definitions and implementations – and Scala objects can be used to instantiate traits. This section presents the fundamental features of multitier modules, how they can be defined to encapsulate a distributed application, and how they support interfaces and different implementations.

4.1.1 Module Definition

In ScalaLoci, multitier modules are defined by a trait with the `@multitier` annotation (cf. Section 3.3 on page 63). Multitier modules can define (i) values placed on different peers and (ii) the peers on which the values are placed – including constraints on the architectural relation between peers. This approach enables modularization across peers (not necessarily along peer boundaries) combining the benefits of multitier programming and modular development. As an example, we consider an application that allows a user to edit documents offline but also offers the possibility to backup the data to an online storage:

```
1 @multitier trait Editor {
2   @peer type Client <: { type Tie <: Single[Server] }
3   @peer type Server <: { type Tie <: Single[Client] }
4
5   val backup: FileBackup
6 }
```

The `Editor` specifies a `Client` (Line 2) and a `Server` (Line 3). The client should be able to backup/restore documents to/from the server, e.g., the client can invoke a `backup.store` method to backup data. Thus, the module requires an instance of the `FileBackup` multitier module (Line 5) providing a backup service. Section 4.2 on page 79 shows how the `Editor` and the `FileBackup` module can be composed.

4.1.2 Encapsulation with Multitier Modules

ScalaLoci's multitier modules *encapsulate distributed (sub)systems* with a specified distributed architecture, enabling the creation of larger distributed applications by composition. Listing 4.1 on the facing page shows a possible implementation for a backup service subsystem using the file system to store backups. The multitier `FileBackup` module specifies a `Processor` to process data (of type `Data`) and a `Storage` peer to store and retrieve data associating them to an ID. The `store` (Line 5) and `load` (Line 7) methods can be called on the `Processor` peer, invoking the `writeFile` method (Line 6) and the `readFile` method (Line 8) remotely on the `Storage` peer. The methods then operate on `Storage` peer's file system.

Listing 4.1 File backup.

```
@multitier trait FileBackup {
  @peer type Processor <: { type Tie <: Single[Storage] }
  @peer type Storage <: { type Tie <: Single[Processor] }

  def store(id: Long, data: Data): Unit on Processor =
    on[Storage].run.capture(id, data) { writeFile(data, s"/storage/$id") }
  def load(id: Long): Future[Data] on Processor =
    on[Storage].run.capture(id) { readFile[Data](s"/storage/$id") }.asLocal
}
```

Overall, the `FileBackup` module encapsulates all the functionalities related to the backup service subsystem, including the communication between `Processor` and `Storage`. ScalaLoci multitier modules further support standard access modifiers for placed values (e.g., `private`, `protected` etc.), which are used as a technique to encapsulate module functionality.

As the last example demonstrates, multitier modules enable separating modularization and distribution concerns, allowing developers to organize applications based on logical units instead of network boundaries. A multitier module abstracts over potentially multiple components and the communication between them, specifying distribution by expressing the placement of a computation on a peer in its type. Both axes are traditionally intertwined by having to implement a component of the distributed system in a module (e.g., a class, an actor, etc.) leading to cross-host functionality being scattered over multiple modules.

4.1.3 Multitier Modules as Interfaces and Implementations

To decouple the code that uses a multitier module from the concrete implementation of such a module, ScalaLoci supports modules to be used as interfaces and implementations. Multitier modules can be abstract, i.e., defining only abstract members, acting as module interfaces, or they can define concrete implementations. For example, applications that require a backup service can be developed against the `BackupService` module interface, which declares a `store` and a `load` method:

```
@multitier trait BackupService {
  @peer type Processor <: { type Tie <: Single[Storage] }
  @peer type Storage <: { type Tie <: Single[Processor] }

  def store(id: Long, data: Data): Unit on Processor
  def load(id: Long): Future[Data] on Processor
}
```

Defining `BackupService` as a multitier module in ScalaLoci allows for specifying the distributed components and the architectural scheme of the subsystem. We use

Listing 4.2 Volatile backup.

```
1 @multitier trait VolatileBackup extends BackupService {
2   def store(id: Long, data: Data): Unit on Processor =
3     on[Storage].run.capture(id, data) { storage += id -> data }
4   def load(id: Long): Future[Data] on Processor =
5     on[Storage].run.capture(id) { storage(id) }.asLocal
6
7   private val storage: Local[Map[Long, Data]] on Storage =
8     Map.empty[Long, Data]
9 }
```

ScalaLoci's peer specification to specify the distributed components (cf. Section 3.2.1 on page 55) on which code is executed.

ScalaLoci adopts Scala's inheritance mechanism to express the relation between the multitier modules used as interfaces and their implementations. The `FileBackup` module of Listing 4.1 on the preceding page is a possible implementation for the `BackupService` module interface, i.e., we can redefine `FileBackup` to let it implement `BackupService`:

```
1 @multitier trait FileBackup extends BackupService {
2   /* ... */
3 }
```

Listing 4.2 presents a different implementation for the `BackupService` module interface. The `VolatileBackup` trait encapsulates the functionalities related to the backup service subsystem, e.g., the `storage` map is declared private (Line 7), so modules that mix in this trait or use instances cannot directly access it. On the distribution axis, `storage` is marked `Local`, i.e., it is not accessible remotely. The implementations of the `store` and of the `load` methods execute remote blocks on the `Storage` peer to access the `storage` map (Lines 3 and 5).

An implementation for the `BackupService` module interface using a database backend as storage is presented in Listing 4.3 on the facing page. The implementations of the `store` and of the `load` methods insert the backup data into a database and retrieve the data from a database, respectively, remotely on the `Storage` peer. The example uses the Quill [Ioffe 2015] query language to access the database (Lines 4 and 9).

4.1.4 Combining Multitier Modules

Thanks to the separation between module interfaces and module implementations, applications can be developed against the interface, remaining agnostic to the

Listing 4.3 Database backup.

```
1  @multitier trait DatabaseBackup extends BackupService {
2    def store(id: Long, data: Data): Unit on Processor =
3      on[Storage].run.capture(id, data) {
4        db.run(query[(Long, Data)].insert(lift(id -> data)))
5      }
6
7    def load(id: Long): Future[Data] on Processor =
8      on[Storage].run.capture(id) {
9        db.run(query[(Long, Data)].filter { _._1 == lift(id) }) map { _.head._2 }
10     }.asLocal
11
12   private val db: AsyncContext = /* ... */
13 }
```

implementation details of a subsystem encapsulated in a multitier module. For example, the `Editor` presented before can be adapted to use a `BackupService` interface instead of the concrete `FileBackup` implementation (Line 5):

```
1  @multitier trait Editor {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4
5    val backup: BackupService
6  }
```

Finally, a multitier module can be instantiated by instantiating concrete implementations of the module interfaces to which it refers. ScalaLoci relies on the Scala approach of using an `object` to instantiate a module, i.e., declaring an object that extends a trait – or mixes together multiple traits – creates an instance of those traits. For example, the following code creates an `editor` instance of the `Editor` module by providing a concrete `DatabaseBackup` instance for the abstract `backup` value:

```
1  @multitier object editor extends Editor {
2    @multitier object backup extends DatabaseBackup
3  }
```

The multitier module instance of a `@multitier object` can be used to run different peers from (non-multitier) standard Scala code (e.g., the `Client` and the `Server` peer), where every peer instance only contains the values placed on the respective peer. Peer startup is presented in Section 3.5 on page 70.

4.2 Abstract Peer Types

In the previous section, we have shown how to encapsulate a subsystem within a multitier module and how to define a module interface such that multiple im-

plementations are possible. ScalaLoci modules allow for going further, enabling abstraction over placement using abstract peer types. Abstract peer types allow a module not only to specify the values and computations for a certain functionality but also to specify the places that are involved in providing and executing such functionality. Consequently, multitier modules can abstract over distributed functionalities.

Peer types are abstract type members of traits, i.e., they can be overridden in sub traits, specializing their type. As a consequence, ScalaLoci multitier modules are parametric on peer types. For example, the `BackupService` module of the previous section defines an abstract `Processor` peer, but the `Processor` peer does not necessarily need to refer to a *physical* peer in the system. Instead, it denotes a *logical* place. When running the distributed system, a `Client` peer, for example, may adopt the `Processor` role, by specializing the `Client` peer to be a `Processor` peer.

Peer types are used to distinguish places only at the type level, i.e., the placement type `T on P` represents a run time value of type `T`. The peer type `P` is used to keep track of the value's placement, but a value of type `P` is never constructed at run time. Hence, `T on P` is essentially a "phantom type" [Cheney and Hinze 2003] due to its parameter `P`.

The next two sections describe the interaction of abstract peer types with two composition mechanisms for multitier modules. We already encountered the first mechanism, module references, which we describe in more details in Section 4.2.1. The other mechanism, multitier mixing, which we describe in Section 4.2.2 on the next page, enables combining multitier modules directly. In both cases, the peers defined in a module can be specialized with the roles of other modules' peers.

4.2.1 Peer Type Specialization with Module References

Since peer types are abstract, they can be specialized by narrowing their upper type bound, augmenting peers with different roles defined by other peers. Peers can subsume the roles of other peers – similar to subtyping on classes – enabling polymorphic usage of peers. Programmers can use this feature to augment peer types with roles defined by other peer types by establishing a subtyping relation between both peers. This mechanism enables developers to define reusable patterns of interaction among peers that can be specialized later to any of the existing peers of an application.

For example, the editor application that requires the backup service (Section 4.1 on page 75) needs to specialize its `Client` peer to be a `Processor` peer and its

Server peer to be a Storage peer for clients to be able to perform backups on the server:

```
1 @multitier trait Editor {
2   @peer type Client <: backup.Processor {
3     type Tie <: Single[Server] with Single[backup.Storage] }
4   @peer type Server <: backup.Storage {
5     type Tie <: Single[Client] with Single[backup.Processor] }
6
7   val backup: BackupService
8 }
```

We specify the Client peer to be a (subtype of the) backup.Processor peer (Line 3) and the Server peer to be a (subtype of the) backup.Storage peer (Line 5). Both backup.Processor and backup.Storage refer to the peer types defined on the BackupService instance referenced by backup. We can use such *module references* to refer to (path-dependent) peer types through a reference to the multitier module.

Since the subtyping relation Server <: backup.Storage specifies that a server *is a* storage peer, the backup functionality (i.e., all values and methods placed on the Storage peer) are also placed on the Server peer. Super peer definitions are locally available on sub peers, making peers composable using subtyping. Abstract peer types specify such subtyping relation by declaring an upper type bound. When augmenting the server with the storage functionality using subtyping, the Tie type also has to be a subtype of the backup.Storage peer's Tie type. This type level encoding of the architectural relations among peers enables the Scala compiler to check that the combined architecture of the system complies to the architectural constraints of every subsystem.

Note that, for the current example, one may expect to unify the Server and the Storage peer, so that they refer to the same peer, specifying type equality instead of a subtyping relation:

```
1 @peer type Server = backup.Storage { type Tie <: Single[Client] }
```

Since peer types, however, are never instantiated – they are used only as phantom types to keep track of placement at the type level – we can always keep peer types abstract, only specifying an upper type bound. Hence, it is sufficient to specialize Server to be a backup.Storage, keeping the Server peer abstract for potential further specialization.

4.2.2 Peer Type Specialization with Multitier Mixing

The previous section shows how peer types can be specialized when referring to modules through *module references*. This section presents a different composition mechanism based on composing traits – similar to *mixin composition* [Bracha and

Listing 4.4 Multiple-master–worker.

```
1 @multitier trait MultipleMasterWorker[T] {
2   @peer type Master <: { type Tie <: Multiple[Worker] }
3   @peer type Worker <: { type Tie <: Single[Master] }
4
5   def run(task: Task[T]): Future[T] on Master =
6     on(selectWorker()).run.capture(task) { task.process() }.asLocal
7 }
```

Cook 1990]. Since ScalaLoci multitier modules can encapsulate distributed subsystems (Section 4.1 on page 75), mixing multitier modules enables including the implementations of different subsystems into a single module.

ScalaLoci separates modules from peers, i.e., mixing modules does not equate to unify the peers they define. Hence, we need a way to coalesce different peers. We use (i) subtyping and (ii) overriding of abstract types as a mechanism to specify that a peer also comprises the placed values of (i) the super peers and (ii) the overridden peers, i.e., a peer subsumes the functionalities of its super peers (Section 4.2.1 on page 80) and its overridden peers. Since peers are abstract type members, they can be overridden in sub modules. Further, overriding peers can specify additional super peers as upper bound for the peer type. To demonstrate mixing of multitier modules we consider the case of two different functionalities.

First, we consider a computing scheme where a master offloads tasks to worker nodes (Listing 4.4). The example defines a master that has a multiple tie to workers (Line 2) and a worker that has a single tie to a master (Line 3). The `run` method has the placement type `Future[T] on Master` (Line 5), placing `run` on the `Master` peer. The `Task` type is parametrized over the type `T` of the value, which a task produces after execution. Running a task remotely results in a `Future` [Baker and Hewitt 1977] to account for processing time and network delays and potential failures. The remote block (Line 6) is executed on the worker, which starts processing the task. The remote result is transferred back to the master as `Future[T]` using `asLocal` (Line 6). A *single* worker instance in a pool of workers is selected for processing the task via the `selectWorker` method (Line 6). The implementation of `selectWorker` is omitted for simplicity.

Second, we consider the case of monitoring (Listing 4.5 on the facing page). a functionality that is required in many distributed applications to react to possible failures [Massie et al. 2004]. The example defines a heartbeat mechanism across a `Monitored` and a `Monitor` peer in a multitier module. The module defines the architecture with a single monitor and multiple monitored peers (Lines 2 and 3). The `monitoredTimedOut` method (Line 5) is invoked by `Monitoring` implementations when a heartbeat was not received from a monitored peer instance for some time.

Listing 4.5 Monitoring.

```
1  @multitier trait Monitoring {
2    @peer type Monitor <: { type Tie <: Multiple[Monitored] }
3    @peer type Monitored <: { type Tie <: Single[Monitor] }
4
5    def monitoredTimedOut(monitored: Remote[Monitored]): Unit on Monitor
6  }
```

Listing 4.6 on the next page provides a possible implementation for the `Monitoring` module: The `MonitoringImpl` module sends a heartbeat every two seconds by remotely invoking the `heartbeat` method (Line 6), which stores the time stamp of receiving the last heartbeat for every monitored instance into a map (Line 10). Note that `heartbeat` is defined as a subjective placed method (`sbj` modifier in Line 9, cf. Section 3.2.7 on page 61), i.e., the `monitored` identifier is bound to a reference to the peer instance that invokes `heartbeat` remotely. The monitor checks the map regularly and invokes the `monitoredTimedOut` method if heartbeats were missing for at least four seconds for a monitored instance (Line 16).

To add monitoring to an application, such application has to be mixed with the `Monitoring` module. Mixing composition brings the members declared in all mixed-in modules into the local scope of the module that mixes in the other modules, i.e., all peer types of the mixed-in modules are in scope. However, the peer types of different modules define separate architectures, which can then be combined by specializing the peers of one module to the peers of other modules. For example, to add monitoring to the the `MultipleMasterWorker` functionality, `MultipleMasterWorker` needs to be mixed with `Monitoring` and the `Master` and `Worker` peers need to be overridden to be (subtypes of) `Monitor` and `Monitored` peers:

```
1  @multitier trait MonitoredMasterWorker[T] extends
2      MultipleMasterWorker[T] with Monitoring {
3
4    @peer type Master <: Monitor {
5      type Tie <: Multiple[Worker] with Multiple[Monitored] }
6    @peer type Worker <: Monitored {
7      type Tie <: Single[Master] with Single[Monitor] }
8  }
```

Specializing peers of mixed modules follows the same approach as specializing peers accessible through module references (Section 4.2.1 on page 80), i.e., `Master <: Monitor` specifies that a master is a monitor peer, augmenting the master with the monitor functionality. Also, for specialization using peers of mixed-in modules, the compiler checks that the combined architecture of the system complies to the architectural constraints of every subsystem.

Listing 4.6 Monitoring implementation.

```
1  @multitier trait MonitoringImpl extends Monitoring {
2    private val timer = new Timer
3    private val monitoredTime = mutable.Map.empty[Remote[Monitored], Long]
4
5    on[Monitored] {
6      timer.schedule(() => remote call heartbeat(), 2000L, 2000L)
7    }
8
9    private def heartbeat() = on[Monitor] sbj { monitored: Remote[Monitored] =>
10     monitoredTime += monitored -> System.currentTimeMillis
11   }
12
13   on[Monitor] {
14     timer.schedule(() => monitoredTime foreach { case (monitored, time) =>
15       if (System.currentTimeMillis - time > 4000L)
16         monitoredTimedOut(monitored)
17     }, 2000L, 2000L)
18   }
19 }
```

4.2.3 Properties of Abstract Peer Types

ScalaLoci abstract peer types share commonalities with both parametric polymorphism – considering type parameters as type members [Odersky et al. 2016; Thorup 1997] – like ML parameterized types [Milner et al. 1975] or Java generics [Bracha et al. 1998], as well as subtyping in object-oriented languages. Similar to parametric polymorphism, abstract peer types allow parametric usage of peer types as shown for the `BackupService` module defining a `Storage` peer parameter. Distinctive from parametric polymorphism, however, with abstract peer types, peer parameters remain abstract, i.e., specializing peers does not unify peer types. Instead, similar to subtyping, specializing peers establishes an *is-a* relation.

Placement types `T on P` support suptyping between peers by being covariant in the type of the placed value and contravariant in the peer, i.e., the `on` type is defined as `type on[+T, -P]`, which allows values to be used in a context where a value of a super type placed on a sub peer is expected. This encoding is sound since a subtype can be used where a super type is expected and values placed on super peers are available on all sub peers. For example, Listing 4.7 on the facing page extends the `Editor` with a `WebClient`, which is a special kind of client (i.e., `WebClient <: Client`, Line 5) with a Web user interface (Line 8), and a `MobileClient` (i.e., Line 6):

By using subtyping on peer types, not unifying the types, we are able to distinguish between the general `Client` peer, which can have different specializations (e.g., `WebClient` and `MobileClient`), i.e., every Web client is a client but not every

Listing 4.7 Editor.

```
1  @multitier trait Editor {
2    @peer type Server <: { type Tie <: Multiple[Client] }
3    @peer type Client <: { type Tie <: Single[Server] }
4
5    @peer type WebClient <: Client { type Tie <: Single[Server] }
6    @peer type MobileClient <: Client { type Tie <: Single[Server] }
7
8    val webUI: UI on WebClient
9    val ui: UI on Client = webUI // ✗ Error: `Client` not a subtype of `WebClient`
10 }
```

client is a Web client. By keeping the types distinguishable, the `ui` binding (Line 9) is rejected by the compiler since it defines a value on the `Client` peer, i.e., the access to `webUI` inside the placed expression is computed on the `Client` peer. However, `webUI` is not available on `Client` since it is placed on `WebClient` and a client is not necessarily a Web client.

4.3 Constrained Multitier Modules

ScalaLoci multitier modules not only allow abstraction over placement but also the definition of *constrained multitier modules* that refer to other modules. This feature enables expressing constraints among the modules of a system, such as that one functionality is required to enable another. Instantiating a constrained module requires a module implementation for each required module. In ScalaLoci, Scala's self-type annotations express such constraints, indicating which other module is required during mixin composition. To improve decoupling, constraints are often defined on module interfaces, such that multiple module implementations are possible.

Applications requiring constrained modules include distributed algorithms, discussed in more detail in the evaluation (Section 7.4.1 on page 135). For example, a global locking scheme ensuring mutual exclusion for a shared resource can be implemented based on a central coordinator. Choosing a coordinator among connected peers requires a leader election algorithm. The `MutualExclusion` module declares a `lock` (Line 2) and an `unlock` (Line 3) method for regulating access to a shared resource. `MutualExclusion` is constrained over `LeaderElection` since our locking scheme requires the leader election functionality:

```
1  @multitier trait MutualExclusion { this: LeaderElection =>
2    def lock(id: T): Boolean on Node
3    def unlock(id: Id): Unit on Node
4  }
```

We express such requirement as a Scala self-type (Line 1), which forces the developer to mix in an implementation of the `LeaderElection` module to create instances of the `MutualExclusion` module.

A leader election algorithm can be defined by the following module interface:

```
1 @multitier trait LeaderElection[T] {
2   @peer type Node
3
4   def electLeader(): Unit on Node
5   def electedAsLeader(): Unit on Node
6 }
```

The module defines an `electLeader` method (Line 4) to initiate the leader election. The `electedAsLeader` method (Line 5) is called by `LeaderElection` module implementations on the peer instance that has been elected to be the leader.

All definitions of the `LeaderElection` module required by the self-type annotation are available in the local scope of the `MutualExclusion` module, which includes peer types and placed values. A self-type expresses a requirement but not a subtyping relation, i.e., we express the requirement on `LeaderElection` in the example as self-type since the `MutualExclusion` functionality requires leader election but is not a leader election module itself. In contrast, using an `extends` clause, i.e., `MutualExclusion extends LeaderElection`, the `LeaderElection` module's definitions become part of the `MutualExclusion` interface.

Multiple constraints can be expressed by using a compound type. For example, different peer instances often need to have unique identifiers to distinguish among them. Assuming an `Id` module provides such mechanism, a module which requires both the leader election and the identification functionality can specify both required modules as compound self-type `this: LeaderElection with Id`. Such requirement makes the definitions of both the `LeaderElection` and the `Id` module available in the module's local scope and forces the developer to mix in implementations for both modules.

Mixin composition is guaranteed by the compiler to conform to the self-type – which is the essence of the Scala *cake pattern*. Assuming a `YoYo` implementation of the `LeaderElection` interface which implements the Yo-Yo algorithm [Santoro 2006] – Section 7.4.1 on page 135 presents different leader election implementations – the following code shows how a `MutualExclusion` instance can be created by mixing together `MutualExclusion` and `YoYo`:

```
1 @multitier object mutualExclusion extends MutualExclusion with YoYo
```

The `YoYo` module implementation of the `LeaderElection` module interface satisfies the `MutualExclusion` module's self-type constraint on the `LeaderElection`

interface. Since mixing together `MutualExclusion` and `YoYo` fulfills all constraints and leaves no values abstract, the module can be instantiated.

4.4 Abstract Architectures

The peer types presented in Section 4.2 on page 79 are abstract but the upper type bound already constrains the architecture involving them. ScalaLoci multitier modules also enable *abstracting over the architectural scheme* that involves some peers. Declaring peers completely abstract without any restriction enables the definition of module interfaces that are polymorphic in the architecture. This feature is important to develop a module that provides a functionality which is applicable to several architectures (such as logging, monitoring, backup, etc.) without tightening the modules to a specific architecture. An interface that is architecture-polymorphic can have specialized implementations for different architectures.

4.4.1 Multitier Modules with Abstract Architectures

ScalaLoci allows the definition of multitier module interfaces that leave the architecture completely abstract. The following example provides a module interface for a computing scheme where a master node offloads work to worker nodes (keeping only the interface of the `MultipleMasterWorker` module of Listing 4.4 on page 82):

```
1 @multitier trait MasterWorker[T] {
2   @peer type Master
3   @peer type Worker
4
5   def run(task: Task[T]): Future[T] on Master
6 }
```

Crucially, the module does not specify any restriction on the architecture, i.e., the peer types `Master` and `Worker` are abstract *and* are not bound to architectural constraints through a `Tie` specification. Implementations of the module can extend the interface and specify any kind of architecture, i.e., the module interface is agnostic to the architectural scheme and can be instantiated to different architectures.

Developers can compose multitier modules that leave (parts of) the architecture abstract. Thus, the combination of modules is parametrized over the architecture, enabling developers to define increasingly complex (sub)systems where the concrete architecture can be fully refined later. In the following examples, we define a client–server architecture where the `MasterWorker` subsystem encapsulates the allocation of computing resources for processing data among the server and the clients. The

module specializes the `Client` and the `Server` peer to the peers of an abstract `MasterWorker` instance, keeping the architecture of the subsystem abstract.

In the first case we specialize the clients to be workers and the server to be a master:

```
1 @multitier trait VolunteerProcessing {
2   @peer type Client <: comp.Worker
3   @peer type Server <: comp.Master
4
5   val comp: MasterWorker[Int]
6
7   on[Server] { comp.run(new Task()) }
8 }
```

In this case, the server can offload work to the connected clients (Line 7), e.g., as a form of volunteer computing where clients donate their computing resources to a project that needs to process large amounts of data, such as infrastructures for computing new prime numbers in the Great Internet Mersenne Prime Search (GIMPS) project.

In the second case, we specialize the clients to be masters and the server to be a worker, i.e., resource-limited clients can offload the processing of the data they collected to a central server with more computational resources (Line 7), like in build servers:

```
1 @multitier trait CentralProcessing {
2   @peer type Client <: comp.Master
3   @peer type Server <: comp.Worker
4
5   val comp: MasterWorker[Int]
6
7   on[Client] { comp.run(new Task()) }
8 }
```

Note that we simply switch the master and worker roles of the client and the server, but we keep the architecture abstract for the implemented subsystem, which allows the subsystem to be used with applications based on different architectural schemes. We define the concrete architectural relations among the peers only when composing the final application.

4.4.2 Implementations for Concrete Architectures

Modules with abstract architectures eventually require module implementations that define concrete architectures. Module interfaces abstracting over architectures allow different module implementations to implement a functionality in a way that is specific to a particular concrete architecture. For example, the

Listing 4.8 Single-master–worker.

```
1 @multitier trait SingleMasterWorker[T] extends MasterWorker[T] {
2   @peer type Master <: { type Tie <: Single[Worker] }
3   @peer type Worker <: { type Tie <: Single[Master] }
4
5   def run(task: Task[T]): Future[T] on Master =
6     on[Worker].run.capture(task) { task.process() }.asLocal
7 }
```

`SingleMasterWorker` module of Listing 4.8 is a possible implementation for the `MasterWorker` interface for a master and a worker with a mutual single tie (Lines 2 and 3). The remote block, which is executed on the worker (Line 6), starts processing the task. The result is transferred back to the master as `Future[T]` using `asLocal` (Line 6).

The `MultipleMasterWorker` module of Listing 4.4 on page 82 is another possible implementation of the `MasterWorker` module interface for an architecture with a master and multiple workers.

4.4.3 Instantiating Concrete Architectures

For modules which are implemented against interfaces that define abstract architectures, we ensure architecture compliance during module instantiation. To create module instances, a concrete module implementation defining an architecture has to be instantiated. For example, to instantiate the `VolunteerProcessing` module, we create a `MultipleMasterWorker` instance for the `comp` value:

```
1 @multitier object application extends VolunteerProcessing {
2   @multitier object comp extends MultipleMasterWorker[Int]
3 }
```

The architecture of the application percolates from the `MultipleMasterWorker` instance – and does not need to be defined explicitly – which defines a concrete architecture where a single master connects to multiple workers and workers connect to single masters. Hence, the application's architecture can be explicitly specified as:

```
1 @peer type Client <: comp.Worker {
2   type Tie <: Single[Server] with Single[comp.Master] }
3 @peer type Server <: comp.Master {
4   type Tie <: Multiple[Client] with Multiple[comp.Worker] }
```

The following code shows a functionally equivalent alternative version of the `VolunteerProcessing` module that is implemented as a *constrained multitier module* (Section 4.2.2 on page 81) instead of using *module references* (Section 4.2.1 on

page 80) to refer to the `MasterWorker` module, i.e., the module defines a self-type on `MasterWorker`:

```
@multitier trait VolunteerProcessing { this: MasterWorker[Int] =>
  @peer type Client <: Worker
  @peer type Server <: Master

  on[Server] { run(new Task()) }
}
```

The constrained module can be composed into an application using mixin composition, mixing together the `VolunteerProcessing` and the `MultipleMasterWorker` modules:

```
@multitier object application extends
  VolunteerProcessing with
  MultipleMasterWorker[Int]
```

5 A Formal Model for Placement Types

> *The equations are only the first step. We will be going beyond mathematics.*
>
> — **The Traveler**

In this chapter, we formalize a core calculus for ScalaLoci based on *placement types* that models peers, placement, remote access, remote blocks and reactives (only signals, for simplicity). The formalization describes the core concepts of ScalaLoci and is a basis to prove our system sound regarding static types and placement. We implemented the calculus in Coq [Coq Development Team 2016] and mechanized the proofs.

5.1 Syntax

The syntax is in Listing 5.1 on the next page. Types are denoted by T. Types of values that can be accessed remotely and transmitted over the network are denoted by U. Placement types S (cf. Section 3.2.2 on page 56) are defined based on a numerable set of peer type names $\mathcal{P}$. Since Scala has no native notion of *peers*, we encode peers as Scala type members. $\mathcal{P}$ corresponds to the set of all *peer types* in the embedding. Besides standard terms, t includes remote access (cf. Section 3.2.3 on page 57), reactives (cf. Section 3.2.4 on page 58) and remote blocks (cf. Section 3.2.8 on page 62). Remote access via asLocal is explicitly ascribed with a type S for the accessed value. We model both aggregation over all remote values of connected peer instances (cf. Section 3.2.6 on page 60) – but we do not distinguish syntactically between different variants of asLocal for aggregation – and selecting a specific peer instance using the from operator (cf. Section 3.2.7 on page 61). A program $l = (\mathcal{T}, \mathcal{S}, \mathcal{I}, s)$ consists of the architecture defined via the ties $\mathcal{T}$ (cf. Section 3.2.1 on page 55), the peer types $\mathcal{S}$ for peer instances $\mathcal{I}$ and the definition of placed values, modeled as nested end-terminated s-terms binding t-terms to names. Thus, s-terms express placed bindings and t-terms are placed expressions, which evaluate to the value that is bound. Placement defined by a term s binds an identifier x

Listing 5.1 Syntax.

$$
\begin{array}{lll}
l ::= (\mathcal{T}, \mathcal{S}, \mathcal{I}, s) & & \text{program} \\
s ::= \mathsf{placed}\ x\colon S = t\ \mathsf{in}\ s \mid \mathsf{end} & & \text{placement term} \\
t ::= \lambda x\colon T.\, t \mid t\, t \mid x \mid \mathsf{unit} \mid & & \text{standard term} \\
\quad \mathsf{none\ of}\ T \mid \mathsf{some}\ t \mid \mathsf{nil\ of}\ T \mid \mathsf{cons}\ t\ t \mid & & \\
\quad \mathsf{asLocal}\ x\colon S\ [\ \mathsf{from}\ t\ ]\ \mid & & \text{remote access term} \\
\quad \mathsf{asLocal\ run}\ x\colon T = t\ \mathsf{in}\ t : S\ [\ \mathsf{from}\ t\ ]\ \mid & & \text{remote block term} \\
\quad \mathsf{signal}\ t \mid \mathsf{var}\ t \mid \mathsf{now}\ t \mid \mathsf{set}\ t := t \mid p \mid & & \text{reactive term} \\
\quad r \mid \vartheta \mid \mathsf{asLocal}\ t : S \mid \mathsf{asLocal}\ t : S\ [\ \mathsf{from}\ t\ ] & & \text{intermediate term} \\
v ::= \lambda x\colon T.\, t \mid \mathsf{unit} \mid \mathsf{end} \mid p \mid r \mid \vartheta \mid & & \text{value} \\
\quad \mathsf{none\ of}\ T \mid \mathsf{some}\ v \mid \mathsf{nil\ of}\ T \mid \mathsf{cons}\ v\ v & & \\
T ::= \mathsf{Option}\ T \mid \mathsf{List}\ T \mid \mathsf{Signal}\ T \mid \mathsf{Remote}\ P \mid \mathsf{Unit} \mid & & \text{type} \\
\quad \mathsf{Var}\ T \mid T \to T & & \\
U ::= \mathsf{Option}\ U \mid \mathsf{List}\ U \mid \mathsf{Signal}\ U \mid \mathsf{Remote}\ P \mid \mathsf{Unit} & & \text{transmittable type} \\
S ::= T\ \mathsf{on}\ P & & \text{placement type} \\
P \in \mathcal{P} & & \text{peer} \\
\mathcal{T} : \mathcal{P} \times \mathcal{P} \to \{\mathsf{multiple}, \mathsf{optional}, \mathsf{single}, \mathsf{none}\} & & \text{ties} \\
\mathcal{S} : \mathcal{I} \to \mathcal{P} & & \text{peer instance type} \\
p = \{i\} \subseteq \vartheta \subseteq \mathcal{I} & & \text{peer instance}
\end{array}
$$

of type S to a term t, where S specifies the placement. We consider a fixed set $\mathcal{I}$ of peer instances that can participate in the evaluation of the program and $\mathcal{S}$ a mapping from peer instances to their peer type P. There can be multiple individual peer instances $p \in \mathcal{I}$ of a peer type P (cf. Section 3.2.5 on page 59). A remote peer reference, which is typed as `Remote[P]` in the Scala embedding, is given the type Remote P in the formal development. $\mathcal{T}$ specifies the tie multiplicity of each two peers. We adopt the notation $P_0 \overset{*}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1) = \mathsf{multiple}$, $P_0 \overset{?}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1) = \mathsf{optional}$, $P_0 \overset{1}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1) = \mathsf{single}$, $P_0 \overset{0}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1) = \mathsf{none}$ and $P_0 \leftrightarrow P_1$ iff $\mathcal{T}(P_0, P_1) \neq \mathsf{none}$ and $\mathcal{T}(P_1, P_0) \neq \mathsf{none}$. Ties between two peers $P_0, P_1 \in \mathcal{P}$ are statically defined and directly correspond to ties between *peer types* in the embedding, defined through `type Tie` at the type level. Reactives include

Vars, which can be set explicitly, and Signals. Both can be accessed to read the current value. Syntactic forms that are not part of the program language and arise only in the evaluation are highlighted in gray.

5.2 Dynamic Semantics

We first introduce the auxiliary construct $\mathcal{I}^{[P]}$ used in the rest of the formalization to denote the set of all peer instances of type P, i.e., $\mathcal{I}^{[P]} = \{p \in \mathcal{I} \mid \mathcal{S}(p) = P\}$. Note that $\mathcal{I}^{[P_1]}$ and $\mathcal{I}^{[P_2]}$ are disjoint for two distinct P_1 and P_2. Ties $\mathcal{T}$ are statically known and constrain the run time connections in a system, e.g., there can only be a single connection for a single tie but an arbitrary number of connections for a multiple tie. For simplicity, we do not model dynamic connections, assuming *fixed* connections along the defined ties $\mathcal{T}$. Each peer instance is connected to all remote instances $\mathcal{I}^{[P]}$ for every tied peer type P. We constrain the number of peer instances for every peer type as follows:

Definition 1. For all pairs of peers $P_0, P_1 \in \mathcal{P}$ holds (i) $P_0 \overset{1}{\rightarrowtail} P_1 \implies \left|\mathcal{I}^{[P_1]}\right| = 1$ and (ii) $P_0 \overset{?}{\rightarrowtail} P_1 \implies \left|\mathcal{I}^{[P_1]}\right| \leq 1$.

Listing 5.2 on the following page shows the auxiliary functions to transmit and aggregate over remote values. ζ models remote value transmission from peer instances ϑ of peer type P_1, all of which provide the remote value v of type T. Traditional values, such as options or lists, do not change during transmission. Signals are transmitted by creating a local signal that reevaluates the dependent remote signal remotely on the peer instances on which the remote signal is placed. When accessing a remote value on peer instances of type P_1 from local peer instances of type P_0, the aggregation results either in a list of all remote values, in an option of the remote value or in the remote value directly depending on the tie between P_0 and P_1. φ constructs a term t' that represents the aggregated result and Φ specifies its type.

Reduction rules To preserve the single multitier *flavor* of ScalaLoci, we model the evaluation as a single thread of execution annotating the reduction relation with a set of peer instances ϑ on which an evaluation step takes place, i.e., the reduction step takes place on all peer instances in ϑ. In the Scala implementation, of course, different steps of evaluations could take place on different instances, e.g., instances can make different decisions based on their internal state (the core calculus, however, is pure). Extending the calculus to allow for deviating reduction steps on different peer instances is possible, but it would complicate the core

Listing 5.2 Auxiliary functions ζ for transmission and φ and Φ for aggregation.

$$\begin{aligned}
\zeta(P_1, \vartheta, \mathsf{none\ of}\ T', T) &= \mathsf{none\ of}\ T' \\
\zeta(P_1, \vartheta, \mathsf{nil\ of}\ T', T) &= \mathsf{nil\ of}\ T' \\
\zeta(P_1, \vartheta, \mathsf{unit}, T) &= \mathsf{unit} \\
\zeta(P_1, \vartheta, \vartheta', T) &= \vartheta' \\
\zeta(P_1, \vartheta, \mathsf{some}\, v, \mathsf{Option}\, T) &= \mathsf{some}\ \zeta(P_1, \vartheta, v, T) \\
\zeta(P_1, \vartheta, \mathsf{cons}\, v_0\, v_1, \mathsf{List}\, T) &= \mathsf{cons}\ \zeta(P_1, \vartheta, v_0, T)\ \ \zeta(P_1, \vartheta, v_1, \mathsf{List}\, T) \\
\zeta(P_1, \vartheta, r, \mathsf{Signal}\, T) &= \mathsf{signal}\ \mathsf{asLocal\ run}\ x\colon \mathsf{Unit} = \mathsf{unit\ in\ now}\ r : T\ \mathsf{on}\ P_1\ \mathsf{from}\ \vartheta \\
&\qquad \text{with } x \text{ fresh}
\end{aligned}$$

$$\Phi(P_0, P_1, T) = \begin{cases} \mathsf{List}\, T & \text{for } P_0 \overset{*}{\rightarrowtail} P_1 \\ \mathsf{Option}\, T & \text{for } P_0 \overset{?}{\rightarrowtail} P_1 \\ T & \text{for } P_0 \overset{1}{\rightarrowtail} P_1 \end{cases}$$

A-CONS
$$\frac{P_0 \overset{*}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T) \qquad t = \varphi(P_0, P_1, \vartheta, v, T)}{\varphi(P_0, P_1, p \,\dot\cup\, \vartheta, v, T) = \mathsf{cons}\, t'\, t}$$

A-VALUE
$$\frac{P_0 \overset{1}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T)}{\varphi(P_0, P_1, p, v, T) = t'}$$

A-NIL
$$\frac{P_0 \overset{*}{\rightarrowtail} P_1}{\varphi(P_0, P_1, \varnothing, v, T) = \mathsf{nil\ of}\ T}$$

A-SOME
$$\frac{P_0 \overset{?}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T)}{\varphi(P_0, P_1, p, v, T) = \mathsf{some}\, t'}$$

A-NONE
$$\frac{P_0 \overset{?}{\rightarrowtail} P_1}{\varphi(P_0, P_1, \varnothing, v, T) = \mathsf{none\ of}\ T}$$

calculus without contributing to modeling the core aspects of ScalaLoci, i.e., static placement and type-safe remote access, which are the properties that are subject of our soundness proofs.

The reduction relation $s; \rho \xrightarrow{\vartheta} s'; \rho'$ reduces the placement term s and the reactive system ρ to s' and ρ' taking a single step on a set of peer instances ϑ. The reactive system ρ stores the reactive values created during the execution. More details about ρ are only relevant to the rules dealing with reactive terms (Listing 5.3d on page 96, described below). The reduction relation $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ for a term t placed on the peer instances ϑ of peer type P and a reactive system ρ evaluates to a term t' and a reactive system ρ' by taking a step on a set of peer instances ϑ'. The rules are in Listing 5.3 on the facing page.

Standard The rules in Listing 5.3a on the next page for reducing a term t are standard except that they are extended with the reactive system ρ and the peer instances where the evaluation takes place. E-APP steps on ϑ when evaluating on the peer instances $\vartheta\colon P$. E-CONTEXT evaluates $E[t]$ on ϑ when t steps on ϑ.

Listing 5.3 Operational semantics.

(a) Standard terms.

$$
\begin{aligned}
E ::= {} & [\cdot] \mid E\,t \mid v\,E \mid \mathsf{some}\,E \mid \mathsf{cons}\,E\,t \mid \mathsf{cons}\,v\,E \mid \\
& \mathsf{asLocal}\,t\colon S\,\mathsf{from}\,E \mid \mathsf{asLocal\ run}\,x\colon T = E\,\mathsf{in}\,t : S \mid \\
& \mathsf{asLocal\ run}\,x\colon T = t\,\mathsf{in}\,t : S\,\mathsf{from}\,E \mid \mathsf{asLocal\ run}\,x\colon T = E\,\mathsf{in}\,t : S\,\mathsf{from}\,v \mid \\
& \mathsf{var}\,E \mid \mathsf{now}\,E \mid \mathsf{set}\,E := t \mid \mathsf{set}\,v := E
\end{aligned}
$$

E-CONTEXT

$$\frac{\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'}{\vartheta\colon P \rhd E[t]; \rho \xrightarrow{\vartheta'} E[t']; \rho'}$$

E-APP

$$\frac{}{\vartheta\colon P \rhd \lambda x\colon T.\, t\,v; \rho \xrightarrow{\vartheta} [x \mapsto v]t; \rho}$$

(b) Placement terms.

E-PLACED

$$\frac{\mathcal{I}^{[P]}\colon P \rhd t; \rho \xrightarrow{\vartheta} t'; \rho'}{\mathsf{placed}\,x\colon T\,\mathsf{on}\,P = t\,\mathsf{in}\,s; \rho \xrightarrow{\vartheta} \!\!\!\rightarrow \mathsf{placed}\,x\colon T\,\mathsf{on}\,P = t'\,\mathsf{in}\,s; \rho'}$$

E-PLACEDVAL

$$\frac{}{\mathsf{placed}\,x\colon T\,\mathsf{on}\,P = v\,\mathsf{in}\,s; \rho \xrightarrow{\mathcal{I}} \!\!\!\rightarrow [x \mapsto v]s; \rho}$$

(c) Remote access terms.

E-ASLOCAL

$$\frac{t' = \varphi(P_0, P_1, \mathcal{I}^{[P_1]}, v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\,v\colon T\,\mathsf{on}\,P_1; \rho \xrightarrow{\vartheta} t'; \rho}$$

E-ASLOCALFROM

$$\frac{t' = \zeta(P_1, \vartheta', v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\,v\colon T\,\mathsf{on}\,P_1\,\mathsf{from}\,\vartheta'; \rho \xrightarrow{\vartheta} t'; \rho}$$

E-BLOCK

$$\frac{t' = \zeta(P_0, \vartheta, v, T_0)}{\vartheta\colon P_0 \rhd \mathsf{asLocal\ run}\,x\colon T_0 = v\,\mathsf{in}\,t : T_1\,\mathsf{on}\,P_1; \rho \xrightarrow{\mathcal{I}^{[P_1]}} \mathsf{asLocal}\,[x \mapsto t']t : T_1\,\mathsf{on}\,P_1; \rho}$$

E-BLOCKFROM

$$\frac{t' = \zeta(P_0, \vartheta, v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal\ run}\,x\colon T = v\,\mathsf{in}\,t : S\,\mathsf{from}\,\vartheta'; \rho \xrightarrow{\vartheta'} \mathsf{asLocal}\,[x \mapsto t']t : S\,\mathsf{from}\,\vartheta'; \rho}$$

E-REMOTE

$$\frac{\mathcal{I}^{[P_1]}\colon P_1 \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\,t : T\,\mathsf{on}\,P_1; \rho \xrightarrow{\vartheta'} \mathsf{asLocal}\,t' : T\,\mathsf{on}\,P_1; \rho'}$$

E-REMOTEFROM

$$\frac{\vartheta''\colon P_1 \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\,t : T\,\mathsf{on}\,P_1\,\mathsf{from}\,\vartheta''; \rho \xrightarrow{\vartheta'} \mathsf{asLocal}\,t' : T\,\mathsf{on}\,P_1\,\mathsf{from}\,\vartheta''; \rho'}$$

Listing 5.3 Operational semantics (continued).

(d) Reactive terms.

E-SourceVar
$$\frac{r \notin \mathrm{dom}(\rho)}{\vartheta\colon P_0 \rhd \mathsf{var}\ v; \rho \xrightarrow{\vartheta} r; (\rho, r \mapsto v)}$$

E-Signal
$$\frac{r \notin \mathrm{dom}(\rho)}{\vartheta\colon P_0 \rhd \mathsf{signal}\ t; \rho \xrightarrow{\vartheta} r; (\rho, r \mapsto t)}$$

E-Set
$$\frac{}{\vartheta\colon P_0 \rhd \mathsf{set}\ r := v; \rho \xrightarrow{\vartheta} \mathsf{unit}; [r \mapsto v]\rho}$$

E-Now
$$\frac{t = \rho(r)}{\vartheta\colon P_0 \rhd \mathsf{now}\ r; \rho \xrightarrow{\vartheta} t; \rho}$$

Placement The reduction rules for placement terms s are in Listing 5.3b on the preceding page. A term $\mathsf{placed}\ x\colon T\ \mathsf{on}\ P = t\ \mathsf{in}\ s$ defines a *placed value* by binding the value of t to x in scope of s (cf. Section 3.2.2 on page 56). E-Placed evaluates a placed term t on all instances $\mathcal{I}^{[P]}\colon P$ of the peer type given by the placement type $T\ \mathsf{on}\ P$. The set ϑ denotes the peer instances where the evaluation steps, which is derived from computation rules of the typing derivation (E-PlacedVal, E-App, E-AsLocal, E-AsLocalFrom, E-Block and E-BlockFrom). E-PlacedVal substitutes an evaluated placed value v in all instances $\mathcal{I}$.

Remote access The reduction rules for remote access terms t are in Listing 5.3c on the preceding page. The variants of asLocal model remote access to placed values and remote blocks (cf. Section 3.2.3 on page 57). The program language can use asLocal only to access a placed *value* through its name binding or to run remote blocks. Note that the semantics allows reduction of a *term* under asLocal to support syntactic forms that arise during evaluation (cf. E-AsLocal, E-AsLocalFrom, E-Block and E-BlockFrom). E-AsLocal accesses the remote value v on peer instances of type P_1 from the local instances ϑ of type P_0. The result is an aggregation φ (Listing 5.2 on page 94) over all remote values (cf. Section 3.2.6 on page 60). By assuming that every peer instance is connected to all instances of a tied peer type and Definition 1 on page 93, the values of all peer instances $\mathcal{I}^{[P_1]}$ are aggregated. The evaluation steps on ϑ to provide the aggregated remote value to all instances ϑ. Similarly, E-AsLocalFrom provides the remote value v from the remote instances ϑ' to all local instances ϑ (cf. Section 3.2.7 on page 61). E-Block applies the value v to a remote block t that is computed remotely on the peer instances of type P_1. A remote block term $\mathsf{asLocal}\ \mathsf{run}\ x\colon T_0 = t_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P$ evaluates t_0, binds the result to x in the scope of t_1 and evaluates t_1 remotely on the instances of P (cf. Section 3.2.8 on page 62). The resulting value of the remote evaluation of type T_1 is provided to the invoking peer instances via E-AsLocal. By assuming that every peer instance is connected to all instances of a tied peer type and Definition 1

on page 93, the evaluation steps on all peer instances $\mathcal{I}^{[P_1]}$. E-REMOTE takes a step in a remote block on the peer instances $\mathcal{I}^{[P_1]}$ of type P_1. Similarly, E-BLOCKFROM and E-REMOTEFROM evaluate remote blocks on a single remote instance p.

Reactive system The rules for reactive terms t in Listing 5.3d on the facing page step on the peer instances ϑ where t is placed and model a pull-based reactive system, where reactives are given semantics as store locations in ρ that contain values v for Vars and thunks t for Signals. The pull-based scheme recomputes the value of a signal r and the signals on which r depends upon each access to r. Designing new propagation systems, e.g., push [Maier et al. 2010] push-pull [Elliott 2009], memory-bounded [Krishnaswami et al. 2012], glitch-free [Drechsler et al. 2014] or fair [Cave et al. 2014], is ongoing research. We leave extending our model with such approaches for future work.

5.3 Static Semantics

The type system guarantees that cross-peer access is safe and consistent with the architecture defined through ties $\mathcal{T}$. It rejects programs where remote values are mixed with local values without converting them via asLocal or where a remote value is accessed on a peer that is not tied.

Typing judgment The typing judgment $\Psi; \Delta \vdash s$ states that s is well-typed under Ψ and Δ. The typing judgment $\Psi; \Delta; \Gamma; P \vdash t : T$ for terms t says that t is well-typed under Ψ, Δ and Γ in the peer context P, i.e., the peer on which term t is placed. $\Gamma ::= \varnothing \mid \Gamma, x{:}\, T$ is the typing environment for variables, $\Delta ::= \varnothing \mid \Delta, x{:}\, S$ is the typing environment for placed variables. We require that the name x is distinct from the variables bound by both Γ and Δ, which can always be achieved by α-conversion. Ψ is the typing environment for reactives. It ranges over mappings from reactives to placement types S. The values held by a reactive always have the same type, which is fixed at creation time of the reactive. The typing rules are in Listing 5.4 on the next page.

Standard The typing rules for terms t in Listing 5.4a on the following page are standard except for T-VAR where the type for x is looked up in both Γ and Δ. In the peer context P, the local x – a locally scoped variable in Γ or a value placed on P in Δ – is accessed simply through x.

Placement The typing rules for placement terms s are in Listing 5.4b on the next page. T-PLACED types the term t of type T on P in the peer context P and extends

Listing 5.4 Typing rules.

(a) Standard terms.

T-VAR
$$\frac{x{:}\,T \in \Gamma \ \vee \ x{:}\,T \text{ on } P \in \Delta}{\Psi; \Delta; \Gamma; P \vdash x : T}$$

T-ABS
$$\frac{\Psi; \Delta; \Gamma, x{:}\,T_1; P \vdash t : T_2}{\Psi; \Delta; \Gamma; P \vdash \lambda x{:}\,T_1.\, t : T_1 \to T_2}$$

T-APP
$$\frac{\Psi; \Delta; \Gamma; P \vdash t_1 : T_2 \to T_1 \qquad \Psi; \Delta; \Gamma; P \vdash t_2 : T_2}{\Psi; \Delta; \Gamma; P \vdash t_1\, t_2 : T_1}$$

T-UNIT
$$\frac{}{\Psi; \Delta; \Gamma; P \vdash \mathsf{unit} : \mathsf{Unit}}$$

T-SOME
$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T}{\Psi; \Delta; \Gamma; P \vdash \mathsf{some}\, t : \mathsf{Option}\, T}$$

T-NONE
$$\frac{}{\Psi; \Delta; \Gamma; P \vdash \mathsf{none\ of}\, T : \mathsf{Option}\, T}$$

T-CONS
$$\frac{\Psi; \Delta; \Gamma; P \vdash t_0 : T \qquad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{List}\, T}{\Psi; \Delta; \Gamma; P \vdash \mathsf{cons}\, t_0\, t_1 : \mathsf{List}\, T}$$

T-NIL
$$\frac{}{\Psi; \Delta; \Gamma; P \vdash \mathsf{nil\ of}\, T : \mathsf{List}\, T}$$

(b) Placement terms.

T-PLACED
$$\frac{\Psi; \Delta, x{:}\,T \text{ on } P \vdash s \qquad \Psi; \Delta; \varnothing; P \vdash t : T}{\Psi; \Delta \vdash \mathsf{placed}\, x{:}\,T \text{ on } P = t \text{ in } s}$$

T-END
$$\frac{}{\Psi; \Delta \vdash \mathsf{end}}$$

(c) Remote access terms.

T-PEER
$$\frac{\vartheta \subseteq \mathcal{I}^{[P_1]}}{\Psi; \Delta; \Gamma; P_0 \vdash \vartheta : \mathsf{Remote}\, P_1}$$

T-ASLOCAL
$$\frac{\Psi; \Delta; \varnothing; P_1 \vdash t : U \qquad P_0 \leftrightarrow P_1 \qquad T = \Phi(P_0, P_1, U)}{\Psi; \Delta; \Gamma; P_0 \vdash \mathsf{asLocal}\, t{:}\,U \text{ on } P_1 : T}$$

T-ASLOCALFROM
$$\frac{\Psi; \Delta; \varnothing; P_1 \vdash t_0 : U \qquad P_0 \leftrightarrow P_1 \qquad \Psi; \Delta; \Gamma; P_0 \vdash t_1 : \mathsf{Remote}\, P_1}{\Psi; \Delta; \Gamma; P_0 \vdash \mathsf{asLocal}\, t_0{:}\,U \text{ on } P_1 \text{ from } t_1 : U}$$

T-BLOCK
$$\frac{\Psi; \Delta; \Gamma; P_0 \vdash t_0 : U_0 \qquad \Psi; \Delta; x{:}\,U_0; P_1 \vdash t_1 : U_1 \qquad P_0 \leftrightarrow P_1 \qquad T = \Phi(P_0, P_1, U_1)}{\Psi; \Delta; \Gamma; P_0 \vdash \mathsf{asLocal\ run}\, x = t_0{:}\,U_0 \text{ in } t_1 : U_1 \text{ on } P_1 : T}$$

T-BLOCKFROM
$$\frac{\Psi; \Delta; \Gamma; P_0 \vdash t_0 : U_0 \qquad \Psi; \Delta; x{:}\,U_0; P_1 \vdash t_1 : U_1 \qquad P_0 \leftrightarrow P_1 \qquad \Psi; \Delta; \Gamma; P_0 \vdash t_2 : \mathsf{Remote}\, P_1}{\Psi; \Delta; \Gamma; P_0 \vdash \mathsf{asLocal\ run}\, x = t_0{:}\,U_0 \text{ in } t_1 : U_1 \text{ on } P_1 \text{ from } t_2 : U_1}$$

Listing 5.4 Typing rules (continued).

(d) Reactive terms.

T-REACTIVE
$$\frac{T_0 \text{ on } P = \Psi(r) \qquad T_0 = \mathsf{Signal}\, T_1 \ \vee \ T_0 = \mathsf{Var}\, T_1}{\Psi; \Delta; \Gamma; P \vdash r : T_0}$$

T-SOURCEVAR
$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T}{\Psi; \Delta; \Gamma; P \vdash \mathsf{var}\, t : \mathsf{Var}\, T}$$

T-SIGNAL
$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T}{\Psi; \Delta; \Gamma; P \vdash \mathsf{signal}\, t : \mathsf{Signal}\, T}$$

T-SET
$$\frac{\Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Var}\, T \qquad \Psi; \Delta; \Gamma; P \vdash t_2 : T}{\Psi; \Delta; \Gamma; P \vdash \mathsf{set}\, t_1 := t_2 : \mathsf{Unit}}$$

T-NOW
$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T_0 \qquad T_0 = \mathsf{Signal}\, T_1 \ \vee \ T_0 = \mathsf{Var}\, T_1}{\Psi; \Delta; \Gamma; P \vdash \mathsf{now}\, t : T_1}$$

the environment for placed variables Δ with the type of x. Placed values are modeled as a series of nested s-terms ending in end typed by T-END.

Remote access The typing rules for remote access terms t are in Listing 5.4c on the facing page. T-PEER types peer instances ϑ of peer type defined by P_1. T-ASLOCAL types remote access to a term t placed on peer P_1 in the peer context P_0. The rule ensures that the type ascription U on P_1 is correct for the placed term t by deriving type U for t in the peer context P_1. The rule ensures that P_0 is tied to P_1. Remote access aggregates over remote values (cf. Section 3.2.6 on page 60) and the type of the aggregation is defined by Φ (Listing 5.2 on page 94). Similarly, T-ASLOCALFROM types remote access to the remote instances given by t_1 of type Remote P_1 (cf. Section 3.2.7 on page 61). The rule ensures that t_1 refers to remote peer instances of peer type P_1, where t_0 of type U is placed. T-BLOCK types the application of t_0 to the remote block t_1 on peer P_1. The term t_0 is typed in the context of the local peer P_0. The block t_1 is typed in the context of the remote peer P_1. The environment Γ for typing t_1 consists only of the typing for x to prevent the block from implicitly closing over remote values, i.e., variables must be passed explicitly (cf. Section 3.2.8 on page 62). Similarly, T-BLOCKFROM types a remote bock on a single remote instance.

Reactive system The typing rules for reactive terms t are in Listing 5.4d. The rule T-REACTIVE types a reactive r of the type defined by Ψ. T-SIGNAL types signal expressions and T-SOURCEVAR types Var instantiations. T-SET requires that the term t_1 to be set to a new value is a Var. T-NOW requires that the term t to be read is either a Var or a Signal.

5.4 Type Soundness

Proving type soundness requires some auxiliary definitions and lemmas. First, we show that a transmitted remote value as modeled by ζ (Listing 5.2 on page 94) can be typed on the local peer:

Lemma 1 (Transmission). *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi; \Delta; \Gamma; P_1 \vdash v : U$ and $t = \zeta(P_1, \vartheta, v, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi; \Delta'; \Gamma'; P_0 \vdash t : U$ for any Δ' and Γ'.*

Proof. By induction on v. □

Second, we show that aggregation modeled by φ (Listing 5.2 on page 94) yields the type given by Φ:

Lemma 2 (Aggregation). *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi; \Delta; \Gamma; P_1 \vdash v : U$ and $t = \varphi(P_0, P_1, \vartheta, v, U)$ and $T = \Phi(P_0, P_1, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi; \Delta'; \Gamma'; P_0 \vdash t : T$ for any Δ' and Γ'.*

Proof. By case analysis on the tie multiplicity $P_0 \leftrightarrow P_1$ and, in the case $P_0 \overset{*}{\rightarrowtail} P_1$, by induction on ϑ and the transmission lemma. □

Next, we provide a definition of typability for the reactive system ρ. We denote with $\mathrm{refs}(\rho)$ the set of all reactive references allocated by ρ:

Definition 2. A reactive system ρ is well-typed with respect to typing contexts Δ, Γ and a reactive typing Ψ, written $\Psi; \Delta; \Gamma \vdash \rho$, iff $\mathrm{refs}(\rho) = \mathrm{dom}(\Psi)$ and $\Psi; \Delta; \Gamma; P \vdash \rho(r) : T$ with $\Psi(r) = \mathsf{Var}\, T \,\mathsf{on}\, P$ or $\Psi(r) = \mathsf{Signal}\, T \,\mathsf{on}\, P$ for every $r \in \mathrm{refs}(\rho)$.

We prove type soundness based on the usual notion of progress and preservation [Wright and Felleisen 1994], meaning that well-typed programs do not get stuck during evaluation. The full proofs are in Appendix A.1 on page 193. We first formulate progress and preservation for terms t:

Theorem 1 (Progress on t-terms). *Suppose t is a closed, well-typed term (that is, $\Psi; \varnothing; \varnothing; P \vdash t : T$ for some T, P and Ψ). Then either t is a value or else, for any ϑ and any reactive system ρ such that $\Psi; \varnothing; \varnothing \vdash \rho$, there is some term t', some ϑ' and some reactive system ρ' with $\vartheta : P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$.*

Proof. By induction on the typing derivation and Definition 1 in the case T-AsLocal. □

Theorem 2 (Preservation on t-terms). *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\Psi; \Delta; \Gamma \vdash \rho$ and $\vartheta : P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ with $\vartheta \in \mathcal{I}^{[P]}$, then $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. By induction on the typing derivation and the aggregation lemma in the case T-AsLocal and the transmission lemma in the cases T-AsLocalFrom, T-Block and T-BlockFrom. □

In the induction for the proofs of both progress and preservation, the case T-Var cannot happen since the term is closed (progress theorem) or cannot step (preservation theorem), respectively. In the cases T-Abs, T-Unit, T-None, T-Nil, T-Peer and T-Reactive, the term is a value, which cannot step. The cases T-Some and T-Cons step with E-Context or the term is a value, which cannot step. The case T-App steps with E-Context or E-App. The case T-AsLocal steps with E-Remote or E-AsLocal. The case T-AsLocalFrom steps with E-Context, E-RemoteFrom or E-AsLocalFrom. The case T-Block steps with E-Context or E-Block. The case T-BlockFrom steps with E-Context or E-BlockFrom. The case T-Signal steps with E-Signal. The case T-SourceVar steps with E-Context or E-SourceVar. The case T-Now steps with E-Context or E-Now. The case T-Set steps with E-Context or E-Set.

Based on progress and preservation for terms t, we prove type soundness for whole programs s:

Theorem 3 (Progress on s-terms)**.** *Suppose s is a closed, well-typed term (that is, $\Psi; \varnothing \vdash s$ for some Ψ). Then either s is a value or else, for any reactive system ρ such that $\Psi; \varnothing; \varnothing \vdash \rho$, there is some term s', some ϑ and some reactive system ρ' with $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$.*

Proof. By case analysis on the typing derivation and the progress theorem for t-terms in the case T-Placed. □

Theorem 4 (Preservation on s-terms)**.** *If $\Psi; \Delta \vdash s$ and $\Psi; \Delta; \varnothing \vdash \rho$ and $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$, then $\Psi'; \Delta \vdash s'$ and $\Psi'; \Delta; \varnothing \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. By case analysis on the typing derivation and the preservation theorem for t-terms in the case T-Placed. □

In the case T-End, the term is a value, which cannot step. The case T-Placed steps with E-Placed or E-PlacedVal.

5.5 Placement Soundness

We prove placement soundness for the core calculus. The full proofs are in Appendix A.2 on page 203. We show that we can statically reason about the peer on which code is executed, i.e., that the peer context P in which a term t is type-checked matches the peer type P of the peer instances ϑ on which t is evaluated.

The type system is sound for a placement if the code placed on a peer P is executed on peer instances of peer type P:

Theorem 5 (Static Placement on t-terms). *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ with $\vartheta \subseteq \mathcal{I}^{[P]}$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i\colon P_i' \rhd t_i; \rho_i \xrightarrow{\vartheta''} t_i'; \rho_i'$ holds $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$ and $P_i = P_i'$.*

Proof. By case analysis on the typing derivation for terms t. □

Theorem 6 (Static Placement on s-terms). *If $\Psi; \Delta \vdash s$ and $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i\colon P_i' \rhd t_i; \rho_i \xrightarrow{\vartheta''} t_i'; \rho_i'$ holds $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$ and $P_i = P_i'$.*

Proof. By case analysis on the typing derivation for terms s. □

Further, we prove that remote access is explicit, i.e., it is not possible to compose expressions placed on different peers without explicitly using asLocal:

Theorem 7 (Explicit Remote Access). *If $\Psi; \Delta; \Gamma; P \vdash t : T$, then every subterm t_i of t with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ is either an explicitly accessed remote term (that is, t_r in one of* asLocal $t_r\colon S$, asLocal $t_r\colon S$ from t_f, asLocal run $x\colon T = t_x$ in $t_r : S$ *or* asLocal run $x\colon T = t_x$ in t_r from $t_f : S$*) or $P = P_i$.*

Proof. By case analysis on the typing derivation of terms t. □

6 A Technical Realization

> *The theory is simple [...] but the implementation is very difficult.*
>
> **— Data**

In this chapter, we present the implementation of ScalaLoci. We provide insights into our approach of embedding ScalaLoci abstractions as a domain-specific language into Scala and describe our experiences with using Scala's type level programming features and Scala's macro system to perform extensive AST transformations. Our work on ScalaLoci is also an experiment on Scala's expressiveness in terms of type level programming, macro programming and syntactic flexibility.

6.1 Design Principles

From the ScalaLoci design considerations outlined in Chapter 3 on page 53, we derive the following principles to guide our language implementation:

#1 Support different architectural models Distributed systems exhibit different architectures. Besides common schemes like client–server or a peer-to-peer, developers should be able to freely define the distributed architecture declaratively (Section 6.3 on page 105).

#2 Make remote communication direct and explicit The programmer should be able to execute a remote access by directly accessing values placed on a remote peer. Although remote communication boilerplate code should be reduced to a minimum, remote access should still be syntactically noticeable since it entails potentially costly network communication (Section 6.3.1 on page 106).

#3 Provide static guarantees The language should provide static guarantees whenever possible to catch errors preferably already at compile-time. In particular, access to remote values should be statically checked to conform to the specified distributed architecture and to provide type safety across distributed components (Section 6.3.2 on page 107).

#4 Support declarative cross-host data flows The language should support abstractions for reactive programming for specifying data flows across hosts since distributed applications are often reactive in nature (Section 6.5 on page 120).

#5 Make abstractions for distribution integrate with existing code The new language abstractions for distributed programming should be orthogonal to present language abstractions and integrate properly with existing code. Embedding our abstractions into a host language fosters reusability of existing code (Section 6.4 on page 110).

6.2 Overview of the ScalaLoci Architecture

Figure 6.1 on the facing page provides an overview of the ScalaLoci implementation, which is organized into two projects.

The *communication* project (Section 6.5 on page 120) handles network communication for the generated peer-specific code of the multitier program. Yet, the project can also be used independently of the multitier language as a remote communication library. The project is divided into three packages that provide interfaces for type-safe remote communication over different network protocols. The remote communication mechanisms are not hard-wired into our implementation and can be extended by implementing the interfaces for (i) message passing over different underlying network protocols (*communicators*), (ii) different data serialization schemes (*serializers*) and (iii) different transmission semantics (*transmitters*), e.g., pull-based remote procedure calls and push-based event streams. We implemented support for different network protocols (e.g., TCP and WebSocket), different serializers (e.g., using µPickle [Li 2014] or circe [Brown 2015b] for serialization to JSON) and different reactive systems (e.g., REScala [Salvaneschi et al. 2014b] and Rx [Meijer 2010]). Developers can plug in such implementations as needed for configuring remote communication.

The *language* project provides the `runtime` package implementing the peer instance life cycle of starting and stopping peer instances and dispatching remote accesses using the communication backend. The `language` package contains the encoding of our domain-specific abstractions into Scala and the Scala type system (Section 6.3 on the facing page). The `language.impl` package implements the macro-based code generation (Section 6.4 on page 110).

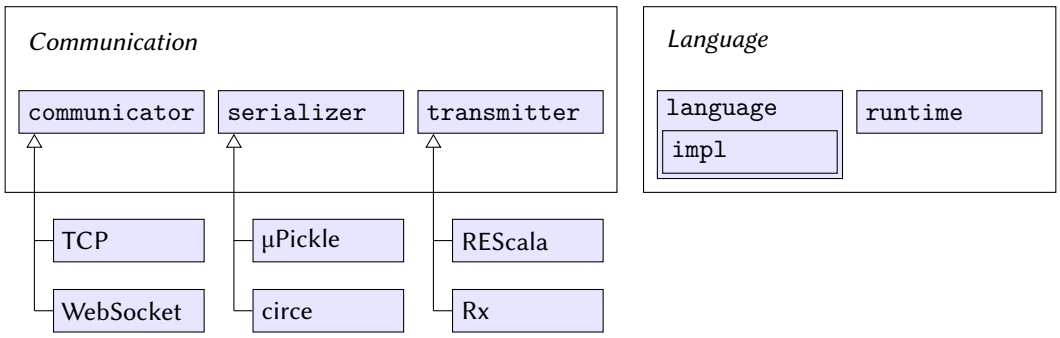


Figure 6.1 Implementation overview.

Cross-platform compilation We support both the standard Scala compiler emitting Java bytecode and the compilation to JavaScript using the Scala.js compiler [Doeraene 2013]. Since there exist libraries which are available only for the JVM or only for JS, multitier modules can mix JVM-only and JS-only libraries if some peers are supposed to run on the JVM and some on a JS virtual machine. Our approach requires a multitier module to be compiled once for every platform on which one of its peers runs. While the implementation of some libraries may not be available to all platforms, the typing information of their definitions is, i.e., the JS compiler can type-check code even if it refers to JVM-only libraries and vice versa. After type-checking and splitting the code, Scala.js' dead code elimination removes all references to JVM libraries, which are not invoked from JavaScript.

6.3 Type Level Encoding

ScalaLoci features the specification of the distributed architecture at the type level by defining (i) the different components as peers and (ii) the topology in which peers are arranged as ties (cf. design principle #1). The type-level encoding solely relies on a combination of standard Scala features, i.e., Scala annotations, abstract type members, type refinements and path-dependent types. For a peer definition `@peer type Registry <: { type Tie <: Multiple[Node] }`, we use valid Scala code instead of deviating more radically from the Scala syntax to provide a more uncluttered definition (e.g., `peer Registry ties Multiple[Node]`). Using standard Scala allows developers to define peers in a syntax with which they are familiar, keeping the appearance of the host language for the domain-specific aspects. Values can be placed on the components defined in the architecture specification and remote access to such values is statically checked by the compiler.

6.3.1 Lexical Context

The following example defines an integer value on the registry and a remote access to the value on the node using `asLocal`. Requiring the `asLocal` syntactic marker makes remote access explicit (cf. design principle #2):

```
1 val i: Int on Registry = 42
2 val j: Future[Int] on Node = i.asLocal
```

Calling `i.asLocal` returns a future of type `Future[Int]`. Knowing from the tie specification that there is a single tie from `Node` to `Registry`, `asLocal` returns a single future. For an optional tie, the `asLocal` call would return an optional future of type `Option[Future[Int]]`. For a multiple tie, `asLocal` would return a sequence of futures.

The example demonstrates the interplay of placement types, peer types and ties when type-checking code (cf. design principle #3). Since the remote access `i.asLocal` happens on the `Node` peer, the value `i` is placed on the `Registry` peer and `Node` defines a single tie to `Registry`, the type system can infer the type of `i.asLocal` to be `Future[Int]`.

When type-checking the `asLocal` call, it is necessary to determine on which peer the call is invoked. For instance, the exact same invocation on the registry peer does not type-check since there is no tie from the `Registry` to the `Registry`:

```
1 val j: Future[Int] on Registry = i.asLocal // ✗ compilation error
```

Thus, typing remote accesses depends on their lexical context. Context information in Scala can be propagated implicitly using implicit values as arguments. The placed expression desugars to a function that takes an implicit argument for the peer context:

```
1 val j: Future[Int] on Node = implicit ctx: Placement.Context[Node] => i.asLocal
```

When type-checking the `i.asLocal` expression, the compiler resolves a value of type `Placement.Context` from the implicit scope, thereby inferring its type parameter which statically captures the peer context. Using implicit functions to propagate context is a common pattern in Scala. Language support for contextual abstractions [Odersky et al. 2017] will be part of Scala 3, allowing for syntactically more lightweight context propagation by omitting the definition of the implicit argument (i.e., `implicit ctx: Placement.Context[Node]`). Since our implementation builds on Scala 2, which does not support such abstractions yet, we use Scala's macro system to synthesize implicit arguments before type-checking (Section 6.4.5 on page 118).

6.3.2 Distributed Architecture

Type-checking remote accesses heavily relies on Scala's type level programming features involving implicit resolution. The interface for accessing a remote value can be freely defined using an *implicit class*, a mechanism to define extension methods for an already defined type. The following code shows the declaration of the `asLocal` method used in the previous examples for placed values of type `V on R` accessed from a local peer `L` resulting in a local value `T` for single ties:

```
1 implicit class BasicSingleAccessor[V, R, T, L](value: V on R)(
2     implicit ev: Transmission[V, R, T, L, Single]) {
3   def asLocal: T = /* ... */
4 }
```

The implementation requires an implicit value of type `Transmission` (Line 2). The implicit `Transmission` value again requires implicit resolution for several values to resolve (i) the current peer context as type parameter `L`, (ii) the tie multiplicity from `L` to `R` (last type parameter) and (iii) the type of the local representation of `V` as type parameter `T`.

The resolution for the current peer context `L` requires an implicit argument of type `Placement.Context[L]` (Section 6.3.1 on the preceding page). After resolving the current peer context `L` – and knowing the peer `R` on which the accessed value is placed by its type `V on R` – the tie from `L` to `R` can be resolved using the scheme presented in Listing 6.1 on the following page. The `single` method (Line 14) resolves a `Tie[L, R, Tie.Single]` specifying a single tie from `L` to `R`. The method implicitly requires a *generalized type constraint* `<:<` (Line 15) which the compiler resolves from the implicit scope if `L` is a subtype of `Any { type Tie <: Single[R] }`, i.e., if the current peer has a `Tie` that is a subtype of `Single[R]`. The resolution for optional and multiple ties is defined analogously (Lines 4 and 9). Letting `TieSingle` inherit from `TieOptional` and `TieOptional` from `TieMultiple` prioritizes the resolution of single ties over optional ties over multiple ties. If no tie can be resolved, peer `L` is not tied to peer `R` and remote access is not possible. It is necessary to find a suitable formulation for determining ties that can be resolved by the Scala compiler since type inference is not specified and implicit search is not guaranteed to be complete. In practice, finding such an encoding requires experimenting with different formulations.

The type of the local representation `T` usually resolves to `Future[V]`, wrapping the accessed value into a future to take network transmission into account. Depending on the concrete data type `T`, other local representations that are more appropriate may be defined. For example, a remote event stream `Event[T]` can be locally represented simply as an `Event[T]` (instead of a `Future[Event[T]]`), which starts propagating events upon remote access. Based on the type, the compiler

Listing 6.1 Tie resolution.

```
sealed trait Tie[L, R, M]

sealed trait TieMultiple {
  implicit def multiple[L, R](
    implicit ev: L <:< Any { type Tie <: Multiple[R] }): Tie[L, R, Tie.Multiple]
}

sealed trait TieOptional extends TieMultiple {
  implicit def optional[L, R](
    implicit ev: L <:< Any { type Tie <: Optional[R] }): Tie[L, R, Tie.Optional]
}

sealed trait TieSingle extends TieOptional {
  implicit def single[L, R](
    implicit ev: L <:< Any { type Tie <: Single[R] }): Tie[L, R, Tie.Single]
}

object Tie extends TieSingle {
  type Single
  type Optional
  type Multiple
}
```

resolves a suitable transmission mechanism from the implicit scope. The resolved transmission mechanism connects the language level to the communication runtime (Section 6.5 on page 120).

In a similar way, variants of `asLocal` for optional and multiple ties return an optional value and a sequence, respectively. Note that we call the `asLocal` variant for accessing remote values on a multiple tie `asLocalFromAll` to make the cost of accessing potentially multiple remote instances visible:

```
implicit class BasicOptionalAccessor[V, R, T, L](value: V on R)(
    implicit ev: Transmission[V, R, T, L, Optional]) {
  def asLocal: Option[T] = /* ... */
}

implicit class BasicMultipleAccessor[V, R, T, L](value: V on R)(
    implicit ev: Transmission[V, R, T, L, Multiple]) {
  def asLocalFromAll: Seq[(Remote[R], T)] = /* ... */
}
```

Encoding the distributed architecture at the type level enables leveraging Scala's expressive type level programming features to type-check remote access based on the type of the accessed value and the architectural relation between the accessing and the accessed peer, guaranteeing static type safety across components.

6.3.3 Lessons Learned

The type level encoding currently adopted in ScalaLoci is a revised version based on our experiences with our prior implementation. Initially, we defined placement types `T on P` as `trait on[T, P]`. Depending on the implicit peer context, our implementation provided an implicit conversion `T on P => T` for local access and `T on P => BasicSingleAccessor` for remote access on a single tie (analogously for optional and multiple ties). We (i) introduced the approach using implicit classes (Section 6.3.2 on page 107) instead of using an implicit conversion for remote access and (ii) defined placed types as type alias `type on[T, P] = Placed[T, P] with T`, i.e., local access does not require an implicit conversion since the compound type directly entails the local representation `T` (and a placement marker `Placed[T, P]`). We decided to remove the need for implicit conversions from our encoding since implicit conversions are only applied if the compiler can infer the target type of the conversion and the compiler does not chain different implicit conversions automatically. Further, reducing the amount of required implicit search improves compile times. The downside of the revised encoding is that a placed value can always be accessed as a local value – even if it is placed on a remote peer. We can, however, reject such illegal access using a check during macro expansion.

Our domain-specific embedding into Scala type-checks valid programs. For rejecting all invalid programs, we employ additional checks when inspecting the typed AST during macro expansion. Over-approximating type correctness in the type level encoding simplifies the encoding. Such approach is especially beneficial when the checks in the macro code are cheaper in terms of compilation performance than Scala's implicit resolution mechanism, which is the case for our approach.

By moving correctness checks to macro code, we reduced the code for the type level encoding from $\sim$600 lines of code in our initial implementation to $\sim$250 lines of code. Issuing compiler errors from macro code also helps in improving error diagnostics since macro code can inspect the AST to give helpful error messages. Debugging compilation issues due to failing implicit resolution, on the other hand, is difficult because the compiler lacks the domain knowledge to hint at which implicit value should have been resolved, resulting in imprecise error messages.

For our purpose of encoding peer and placement types, the key feature of the host language is an expressive type system. Our embedding is based on Scala's unique combination of type system features, namely abstract type members, type refinements, subtyping and path-dependent types. Scala's syntactic flexibility (e.g., writing `T on P` instead of `on[T, P]`) enables an embedding that sorts well with both the host language and the domain-specific abstractions. We conjecture that a similar encoding is possible in languages with similarly expressive type

level programming features – of course relying on the type system features of the host language, which might differ from the Scala features which we use. A Haskell implementation, for example, would neither have to support subtyping, nor could it use such type system feature for the encoding. Any domain-specific embedding always compromises between the domain language and host language characteristics to foster easy integration of orthogonal language features of an existing general purpose language and enabling reuse of existing code.

6.4 Macro Expansion

Compiling multitier programs requires splitting the code into deployable components. In ScalaLoci, peer types define the set of components and placement types of value definitions indicate the components to which the values belong. We use Scala macro annotations to split the code by transforming the AST of a multitier module. Scala macro annotations only expand locally, i.e., they only allow the transformation of the annotated class, trait or object. By local expansion, we retain the same support for separate compilation that Scala offers, enabling modular development of multitier applications. Inside every multitier module (i.e., the annotated class, trait or object), we create a nested trait for every peer type which contains the values placed on the peer. This process automatically conflates the values placed on the same peer without requiring the developer to do so manually, disentangling language-level support for modularization form distribution concerns.

To the best of our knowledge, our approach to code splitting is the most extensive use of Scala macros to date, amounting to $\sim$5.5 K lines of code (compared to $\sim$3.5 K for *Scala Spores* mobile closures [Miller et al. 2014] and the *ScalaTest* testing framework [Venners 2009], $\sim$2 K for *Scala Async* asynchronous programming abstractions [Haller and Zaugg 2012], $\sim$2 K for the *shapeless* generic programming library [Sabin 2011], and $\sim$1 K for the *circe* JSON library [Brown 2015b]). Our implementation confirms that macro-based code generation is a powerful tool for embedding domain-specific abstractions into Scala using compile-time metaprogramming. Crucially, macros run as part of the Scala compiler and have access to type information of the AST, which is important for our use case of splitting code based on peer types.

Scala supports macros in two different flavors, *def macros* expanding expressions and *macro annotations* expanding declarations of classes, traits, objects or members. Hence, to mark a class, trait or object as multitier module, we rely on macro annotations. In contrast to def macros, which expand typed ASTs, macro annotations expand untyped ASTs. AST transformation before type-checking is considered too

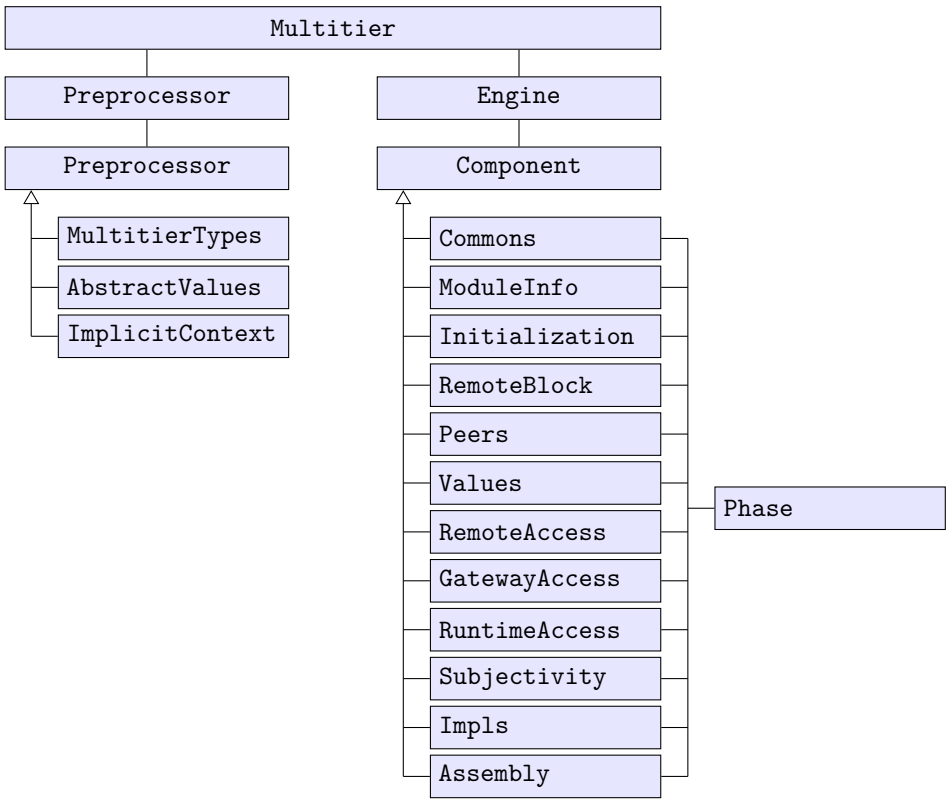


Figure 6.2 Software architecture of the macro expansion.

powerful since it may change Scala's language semantics significantly [Burmako 2013a]. In spirit of keeping our semantics as close as possible to plain Scala, multitier code is syntactically valid and type-correct Scala. Hence, before performing the splitting transformation, we invoke the Scala type checker to obtain a typed AST of the multitier module. Manually invoking the type checker is quite delicate in Scala's current macro system (Section 6.4.6 on page 118).

Our approach allows accessing libraries from the Java, Scala and Scala.js ecosystems from multitier code (cf. design principle #5), even using mixed Scala/Scala.js multitier modules (Section 6.2 on page 104).

6.4.1 Macros Architecture

Figure 6.2 provides an overview of the overall architecture of our macro implementation. The `Multitier` object (at the top) defines the entry point for the macro expansion that is invoked by the Scala compiler to expand `@multitier` macro annotations. The compiler passes the AST of the annotated module to the macro expansion and retrieves the transformed AST as result. We first run a sequence of preprocessing steps (left side) on the untyped AST, compensating for the lack

of contextual abstractions in current Scala (cf. Section 6.4.5 on page 118). Second, we load a set of components (right side) that comprise the actual code generation. Every component defines potentially multiple processing phases, which specify constraints on whether they should run before/after other phases. All phases run sequentially (satisfying their constraints) to transform certain aspects of the AST. A phase involves one or more AST traversals. So far, we did not optimize the code for minimizing traversals to increase compilation performance.

The processing pipeline first splits the multitier module into its top-level definitions (i.e., the members of the annotated class, trait or object), containing the respective sub-AST together with meta information, such as the peer on which a value is placed extracted from its type. The following phases work on this set of definitions. Instead of using a fully-fledged own intermediate representation, we use standard Scala ASTs enriched with additional information, which proved effective for our use case. The final phase assembles the AST of the complete expanded multitier module.

The largest part of the code base deals with the splitting of placed values (`Values` component, Section 6.4.2) and the rewriting of remote accesses from direct style via `asLocal` into calls into the communication backed (`RemoteAccess` component, Section 6.4.3 on page 115), which accounts to almost 2 K lines of code.

6.4.2 Code Splitting Process

The code generation process splits the code according to the placement type of values and creates the necessary run time information for dispatching remote accesses correctly. We consider the following multitier module, which simply defines a single peer `MyPeer` and a single value `i` placed on `MyPeer`:

```
1 @multitier trait SimpleModule {
2   @peer type MyPeer <: { type Tie <: Single[MyPeer] }
3   val i: Int on MyPeer = 1
4 }
```

The macro generates signature values to uniquely identify the multitier module and the peer types it defines and creates a runtime representation of the tie specification:

```
1 @MultitierModule trait SimpleModule {
2   /* ... */
3
4   protected lazy val $loci$mod = "SimpleModule"
5   protected lazy val $loci$sig = Module.Signature($loci$mod)
6   lazy val $loci$peer$sig$MyPeer = Peer.Signature("MyPeer", $loci$sig)
7   val $loci$peer$ties$MyPeer = Map($loci$peer$sig$MyPeer -> Peer.Tie.Single)
8 }
```

The signatures and tie specification are used when setting up connections between peer instances at run time to ensure that the connections conform to the static tie constraints. Further, a `Marshallable` instance – for marshalling and unmarshalling values for network transmission – and a signature for every value is created:

```
1  @MultitierModule trait SimpleModule {
2    /* ... */
3
4    @MarshallableInfo[Int](-660813075)
5    protected final val $loci$mar$SimpleModule$0 = Marshallable[Int]
6
7    @PlacedValueInfo("i:scala.Int", null, $loci$mar$SimpleModule$0)
8    final val $loci$val$SimpleModule$0 = new PlacedValue[Unit, Future[Int]](
9      Value.Signature("i:scala.Int", $loci$mod, $loci$sig.path),
10     Marshallables.unit, $loci$mar$SimpleModule$0)
11 }
```

Line 5 resolves a `Marshallable` instance using Scala's implicit resolution, i.e., it is guaranteed at compile time that a value is serializable and can be accessed over the network. Line 8 defines the run time representation for the placement of value `i`, whose remote access does not take any arguments (type `Unit`) and returns a future (type `Future[Int]`). Line 9 defines the signature of `i` for remote dispatch. Line 10 defines the `Marshallable` instances for the arguments and the return value of `i`.

`Marshallable` instances require a concrete type, for which the concrete serialization format is known at compile time. Since such information is not available for abstract types – e.g., generic type parameters for parameterized modules – the macro expansion defers the `Marshallable` resolution to the specific implementations of the multitier module that define a concrete type for the parameter. The following example shows the `Marshallable` instance (Line 4) for a value of type `T` in a parameterized module (Line 1), which delegates to a method (Line 7) that is to be implemented in a sub-module:

```
1  @MultitierModule trait SomeModule[T] {
2    /* ... */
3
4    protected final val $loci$mar$SomeModule$0 = $loci$mar$deferred$SomeModule$0
5
6    @MarshallableInfo[T](0)
7    protected def $loci$mar$deferred$SomeModule$0: Marshallable[T, T, Future[T]]
8  }
```

As a next step, the macro performs the splitting of placed values (Listing 6.2 on the next page). After expansion, the placed value `i` at the module-level is nulled and annotated to be *compile-time-only* (Line 5), i.e., the value cannot be accessed in

Listing 6.2 Splitting of placed values.

```
1  @MultitierModule trait SimpleModule {
2    /* ... */
3
4    @compileTimeOnly("Remote access must be explicit.") @MultitierStub
5    val i: Int on MyPeer = null.asInstanceOf[Int on MyPeer]
6
7    trait `<placed values of SimpleModule>` extends PlacedValues {
8      val i: Int = $loci$expr$SimpleModule$0()
9      protected def $loci$expr$SimpleModule$0(): Int = null.asInstanceOf[Int]
10   }
11
12   trait $loci$peer$MyPeer extends `<placed values of SimpleModule>` {
13     protected def $loci$expr$SimpleModule$0(): Int = 1
14   }
15 }
```

plain Scala code. The value is kept as a compile-time-only value such that other multitier modules can be type-checked against this module. After type-checking, the macro removes references to compile-time-only values in multitier code. The compile-time-only approach allows us to keep the static specification of placed values (and their placement types) but remove their implementation. Instead, our transformation creates a nested trait for every peer to separate the peer-specific implementations of placed values.

The code generation creates a `<placed values>` trait nested inside the multitier module (Line 7). The trait contains all values of the multitier module. In particular, it defines the placed value `i` (Line 8), which is of type `Int` (instead of type `Int on MyPeer` as the module-level definition), i.e., placement types are erased from generated code on the implementation side. Note that placement types are retained on the specification side (Line 5) for type-checking at compile time. The value `i` is initialized by calling the generated method `$loci$expr$SimpleModule$0` (Line 8), which is nulled by default (Line 9). The value is nulled since it is not available on every peer but we need to keep the value in the `<placed values>` trait to retain Scala's evaluation order. The `MyPeer`-specific `$loci$peer$MyPeer` trait specializes the `<placed values>` trait making for `i` being initialized to `1` for `MyPeer` peer instances (Line 13). When starting a peer, the peer-specific subtraits are instantiated.

Finally, for every generated peer trait, the macro synthesizes a `$loci$dispatch` method, which accesses the placed values for remote accesses (Listing 6.3 on the facing page). For providing a placed value to a remote instance (Line 10), the local value is accessed (Line 11), potentially unmarshalling its arguments and marshalling its return value (Line 12). In case the module does not contain a definition for the value, remote dispatch is delegated to the super module (Line 15), mirroring Scala's method dispatch.

Listing 6.3 Synthesized dispatch method.

```
@MultitierModule trait SimpleModule {
  /* ... */

  trait $loci$peer$MyPeer extends `<placed values of SimpleModule>` {
    /* ... */

    def $loci$dispatch(request: MessageBuffer, signature: Value.Signature,
        reference: Value.Reference) =
      signature match {
        case $loci$val$SimpleModule$0.signature =>
          Try { i } map { response =>
            $loci$mar$SimpleModule$0.marshal(response, reference)
          }
        case _ =>
          super.$loci$dispatch(request, signature, reference)
      }
  }
}
```

6.4.3 Macro Expansion for Remote Access

Since, in our example, we define a `MyPeer`-to-`MyPeer` tie, we can access the value `i` remotely from another `MyPeer` instance. We add the remote access `i.asLocal` (Line 4):

```
@multitier trait SimpleModule {
  @peer type MyPeer <: { type Tie <: Single[MyPeer] }
  val i: Int on MyPeer = 1
  val j: Future[Int] on MyPeer = i.asLocal
}
```

Similar to the expansion of value `i` (cf. Section 6.4.2 on page 112), the definition of value `j` is extracted into a peer-specific method `$loci$expr$SimpleModule$1` (Listing 6.4 on the following page). The transformation from the `i.asLocal` user code to the call into the runtime system (Lines 8–11) ties the knot between the direct-style remote access of ScalaLoci multitier code and the message-passing-based network communication of ScalaLoci's communication backend (Section 6.5 on page 120). The interface for remote access (i.e., the `asLocal` call in the example) is declared by an implicit class. In the example, the interface is defined by the `BasicSingleAccessor` implicit class, which requires an implicit `Transmission` argument for accessing the remote value (cf. Section 6.3.2 on page 107). The `Transmission` argument is rewritten by the macro to a `RemoteRequest` (Line 9) that is instantiated with (i) the arguments for the remote call, (ii) the signature of the accessed value, (iii) the signature of the peer on which the value is placed and (iv) a reference to the runtime

Listing 6.4 Macro expansion for remote access.

```
1  @MultitierModule trait SimpleModule {
2    /* ... */
3
4    trait $loci$peer$MyPeer extends `<placed values of SimpleModule>` {
5      /* ... */
6
7      protected def $loci$expr$SimpleModule$1(): Unit =
8        BasicSingleAccessor[Int, MyPeer, Future[Int], MyPeer](RemoteValue)(
9          new RemoteRequest(
10           (), $loci$val$SimpleModule$0, $loci$peer$sig$MyPeer, $loci$sys)
11         ).asLocal
12   }
13 }
```

system (inherited from `PlacedValues` trait) that manages the network connections. With these information assembled by macro expansion, `asLocal` can perform the remote access.

6.4.4 Macro Expansion for Composed Multitier Modules

Listing 6.5 on the next page illustrates our module composition mechanisms. The ScalaLoci code (Listing 6.5a on the facing page) defines a `ComposedModule` which mixes in the `SimpleModule` from the previous examples (Line 1) and defines a reference to a `SimpleModule` instance (Line 5).

In the generated code (Listing 6.5b on the next page), mixing `SimpleModule` into `ComposedModule` results in the respective `<placed values>` and peer traits being mixed in (Lines 12 and 32) using Scala's mixin composition. For the `inner` reference (Line 5), both the generated module signature (Lines 6–8) and the dispatching logic for remote requests (Lines 23–25) take the path of the module reference (`"inner"`) into account to handle remote access to path-dependent modules. For the `MyPeer` trait of the `ComposedModule` (Line 31), the `inner` reference is instantiated to the `MyPeer` trait of the `SimpleModule` instance `inner` (Line 35), so that values placed on `MyPeer` of the `ComposedModule` can access values placed on `MyPeer` of the `SimpleModule` since `ComposedModule` defines `MyPeer <: inner.MyPeer`. Since peer types are used to guide the splitting and define the composition scheme of the synthesized peer traits, peer types themselves are never instantiated. Hence, they can be abstract.

As illustrated by the example, the code generation solely replaces the code of the annotated trait, class or object and only depends on the super traits and classes and the definitions in the multitier module's body, thus retaining the same support for separate compilation offered by standard Scala traits, classes and objects.

Listing 6.5 Macro expansion for composed multitier modules.

(a) ScalaLoci user code.

```
@multitier trait ComposedModule extends SimpleModule {
  @peer type MyPeer <: inner.MyPeer {
    type Tie <: Single[MyPeer] with Single[inner.MyPeer] }

  @multitier object inner extends SimpleModule
}
```

(b) Generated Scala code after macro expansion.

```
@MultitierModule trait ComposedModule extends SimpleModule {
  @peer type MyPeer <: inner.MyPeer {
    type Tie <: Single[MyPeer] with Single[inner.MyPeer] }

  @MultitierModule object inner extends SimpleModule {
    override protected lazy val $loci$mod = "ComposedModule#inner"
    override protected lazy val $loci$sig = Module.Signature(
      ComposedModule.this.$loci$sig, "inner")
  }

  trait `<placed values of ComposedModule>` extends
      `<placed values of SimpleModule>` with PlacedValues {
    final lazy val inner = $loci$multitier$inner()
    protected def $loci$multitier$inner() =
      new ComposedModule.this.inner.`<placed values of ComposedModule>` { }

    def $loci$dispatch(request: MessageBuffer, signature: Value.Signature,
        reference: Value.Reference) =
      if (signature.path.isEmpty)
        super.$loci$dispatch(request, signature, reference)
      else
        signature.path.head match {
          case "inner" =>
            inner.$loci$dispatch(request,
              signature.copy(path = signature.path.tail), reference)
          case _ =>
            super.$loci$dispatch(request, signature, reference)
        }
  }

  trait $loci$peer$MyPeer extends
      super[SimpleModule].$loci$peer$MyPeer with
      `<placed values of ComposedModule>` {
    protected def $loci$multitier$inner() =
      new ComposedModule.this.inner.$loci$peer$MyPeer { }
  }

  protected lazy val $loci$mod = "ComposedModule"
  protected lazy val $loci$sig = Module.Signature($loci$mod)
  lazy val $loci$peer$sig$MyPeer = Peer.Signature(
    "MyPeer", List(inner.$loci$peer$sig$MyPeer), $loci$sig)
  val $loci$peer$ties$MyPeer = Map(
    $loci$peer$sig$MyPeer -> Peer.Tie.Single,
    inner.$loci$peer$sig$MyPeer -> Peer.Tie.Single)
}
```

6.4.5 Peer Contexts

Before invoking the type checker, the macro performs a transformation step on the untyped AST to compensate for the lack of contextual abstractions in Scala 2, which are to be available Scala 3 [Odersky et al. 2017]. The current context determines for any expression to which peer it belongs (cf. Section 6.3.1 on page 106). Since the context needs to be available to the type checker, the transformation has to take place before type-checking. It transforms placed expressions `e` to `implicit ! => e`, where `!` is the name of the argument carrying the (implicit) peer context. In the lexical scope of the expression `e`, the context can be resolved by the compiler from the implicit scope. For better IDE support, the implicit argument can also be written explicitly by the developer, in which case we do not transform the expression.

6.4.6 Interaction with the Type System

Since (i) we rely on type-checked ASTs for guiding the code splitting by placement types and (ii) splitting changes the shape of the multitier module (i.e., adding members to the annotated module), essentially changing the module's type, the AST transformation needs to be performed during the compilation process. Scala's macro system enables such interaction with the type system, which is essential for splitting ScalaLoci multitier code, in contrast to code generation approaches that run strictly before the compiler.

Yet, in our experience, invoking the type checker for annotated classes, traits or objects is quite fragile with the current Scala macro system. Type-checking the AST again after transformation, where trees are re-type-checked in a different lexical context after transformation, can easily corrupt the owner chain of the compiler's symbol table. To work around those issues, we implemented a transformation that converts ASTs such that they can be re-type-checked. This transformation is independent of the code splitting and is available as a separate project [Weisenburger 2015].

Type-checking multitier modules expands all nested macro invocations. We extensively used ScalaLoci with the REScala domain-specific language for reactive programming [Salvaneschi et al. 2014b] that relies on *def macros* (i.e., expression-based macros). We did not observe any issue with mixing different macro-based language extensions. Invoking the type checker for *macro annotations* (i.e., annotation-based macros) on modules which are themselves nested into other modules, however, is not supported by the current macro system.

6.4.7 Lessons Learned

In our experience, performing complex AST transformations is quite involved using the current Scala macro system, which lacks built-in support for automatic hygiene [Burmako 2013b], i.e., separating the lexical scopes of the macro implementation and the macro call site to guarantee the absence of name clashes between user code and code generated by macro expansion. The macro developer is responsible for ensuring that the generated code does not interfere with the lexical scope of the macro call site by creating identifier names that are expected to be unique for name bindings or using fully qualified names to refer to existing values or types. Moreover, the macro system exposes compiler internals such as the compiler's symbol table, which developers have to keep consistent when transforming typed ASTs. When moving ASTs between different contexts or mixing typed ASTs with newly generated untyped ASTs, developers have to fix the symbol chain manually or re-type-check the AST.

This complex interaction with the type system is the reason why the macro system considered for the next version of Scala (and currently being implemented in the Dotty compiler) does not allow explicit interaction with the type system [Odersky and Stucki 2018]. The new TASTy reflection API properly abstracts over compiler internals and only supports ASTs that are already typed. Macro systems of other languages, such as Racket, are more powerful, supporting the addition of new syntactic forms through the transformation of arbitrary Racket syntax. The revised Scala macros, however, are still more powerful than macro systems like Template Haskell, which do not take the AST as input without explicitly quoting expressions at the call site. Expanding macros on typed ASTs helps in controlling the power of macros and keeping syntactic and semantic deviations from plain Scala small. Prohibiting any kind of interaction with the type system, however, seems too limiting and would make a macro-based implementation of ScalaLoci impossible. Other use cases currently covered by macro annotations are also not well supported under the new scheme, e.g., auto-generation of lenses for manipulating data structures [Trufaut 2014] or auto-generation of serializers [Brown 2015a]. To restore support for such use cases, we could imagine an approach that allows macros to change types in a clearly-defined and controlled way. For instance, the macro could be expanded in several phases that allow for different kinds of modifications:

1. In a first phase, the macro can inspect (but not transform) the current untyped AST and declare members, super traits or super classes and annotations that should be added to the annotated class, trait, object or its companion. Only declarations are required for the following type-checking, not necessarily their definitions.

2. The complete code is type-checked.

3. Similar to the first phase, the macro can inspect (but not transform) the tree which, in contrast to the first phase, is now type-checked. The macro can again declare members, super traits or super classes and annotations that should be added to the annotated class, trait, object or its companion. It may be necessary to further restrict the members, which could be declared, e.g., disallowing adding members that interfere with implicit or method overload resolution since both already happened as part of the type-checking in the second phase.

4. The new member declarations are type-checked. Since no members can be removed and adding members can be restricted appropriately, it is sufficient to only type-check the new members.

5. Finally, macro annotations are expanded. Macro expansion works on type-checked ASTs. Members generated in the previous phases are visible to the macro.

We believe that well-defined interfaces for macros are essential to retain the current level of usefulness of macro annotations in a future macro system while avoiding the issues of the current macro system.

6.5 Runtime

The ScalaLoci communication runtime hides the implementation details of network communication, e.g., data serialization and underlying network protocols, such that developers can access remote values in direct style (via `asLocal`) instead of explicitly sending network messages and registering callbacks for receiving messages. Figure 6.3 on the next page shows the communication runtime which underlies a ScalaLoci multitier program. Our runtime system provides abstraction layers for different network protocols, serialization schemes and the type-safe transmission of values.

6.5.1 Communicators

The lower layer defines *communicators* abstracting over network protocols. We currently support TCP (on the JVM), WebSocket (on the JVM and in web browsers) and WebRTC (in web browsers). Communicators can be instantiated in *listening* mode (e.g., binding a local TCP port and listening for incoming connections) or *connecting* mode (e.g., initiating a TCP connection to a remote host). After establishing

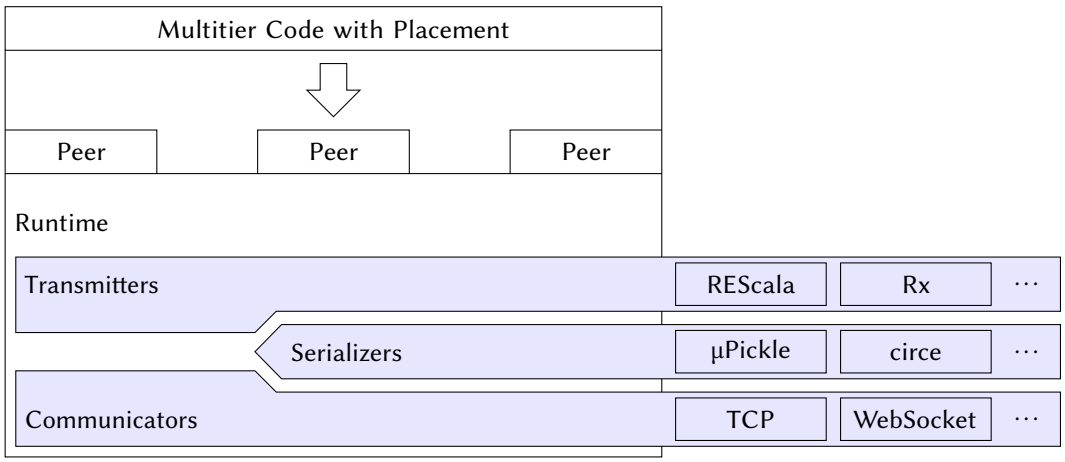


Figure 6.3 Communication runtime.

a connection, the communicators of both endpoints create a `Connection` object that provides a bidirectional message-passing channel, abstracting over the communication model of the underlying protocol, such as TCP byte streams or WebSocket messages. Communicators also offer additional meta information about the established network connection to the higher level, such as if the connection is secure (i.e., encrypted and integrity-protected) or authenticated (and which user token or certificate was used for authentication). Currently, communicators are restricted to point-to-point bidirectional channels. Yet, in the future, we may support additional communication schemes, e.g., broadcasting, single-shot request–response, etc.

6.5.2 Serializers

To serialize values of a specific type for network transmission, the runtime requires an implementation of the `Serializable` type class for every such type – encoded in Scala using the *concept pattern* [Oliveira et al. 2010]. The compiler derives `Serializable` instances using Scala's implicit resolution, guaranteeing that values of a certain type are serializable.

The type class `Serializable[T]` witnesses that a value of type `T` is serializable by providing methods for both serialization and deserialization:

```
1 trait Serializable[T] {
2   def serialize(value: T): MessageBuffer
3   def deserialize(value: MessageBuffer): Try[T]
4 }
```

The runtime invokes the `Serializable` methods to convert between a value of type `T` and a `MessageBuffer`, buffering the byte array to be sent or received over the network. Serialization is not expected to fail, but deserialization may result in a

runtime error if the buffer does not contain a valid serialization. Hence, the return value is wrapped in a `Try`, which represents either a success value or a failure.

We implemented two serializers that simply forward to the µPickle [Li 2014] or circe [Brown 2015b] serialization libraries, respectively. The µPickle serializer, for example, is declared as an implicit value of type `Serializable[T]` given that the compiler is able to resolve implicit instances for `Writer[T]` and `Reader[T]` (which are type classes defined by µPickle for serializing and deserializing values):

```
implicit def upickleSerializable[T]
  (implicit writer: Writer[T], reader: Reader[T]): Serializable[T] = /* ... */
```

For a required implicit value of type `Serializable[Int]`, the compiler automatically resolves the call `upickleSerializable(upickle.default.IntWriter, upickle.default.IntReader)`, constructing a `Serializable[Int]` instance based on µPickle's `IntWriter` and `IntReader`.

6.5.3 Transmitters

The higher level defines *transmitters* implementing the transmission semantics specific to a certain data type. To make a type of value available for transmission, the runtime requires an implementation of the `Transmittable` type class for every such type. A `Transmittable[B, I, R]` instance witnesses that a value of type `T` can be send over the network as value of type `I` and whose local representation after remote access is of type `R`. When accessing a value remotely, the runtime creates related `Transmittable` instances on both connection endpoints.

Primitive and standard collection values are retrieved from a remote instance in a pull-based fashion upon each request. Accessing event streams, on the other hand, does not exhibit pull-based semantics. Instead, events are pushed to remote instances for every event occurrence. Runtime support for accessing event streams remotely is crucial since communication in distributed systems is often event-based [Carzaniga et al. 2001; Meier and Cahill 2002] and event streams allow for data across hosts to be specified in a declarative way (cf. design principle #4). To support such use case, the runtime allows transmitters to operate in *connected* mode, providing a typed message channel between both communication endpoints. The remote event stream uses the channel to propagate events in a push-based manner over the network. The runtime can multiplex multiple such message channels over the same underlying network connection.

The runtime comes with built-in support for transmitting primitive values and standard collections. We further implemented transmitters for REScala [Salvaneschi et al. 2014b] reactives and Rx [Meijer 2010] observables, which developers can plug

in when needed. The runtime is extensible by defining `Transmittable` type class instances for additional types.

Transmitters abstract over different semantics for propagating values to remote hosts. Depending on the type of the value that is accessed remotely, the compiler automatically selects an appropriate transmitter through implicit resolution. The communication runtime performs the actual network communication for remote accesses at the ScalaLoci language level that are transformed into calls into the runtime during macro expansion (cf. Section 6.4.3 on page 115). The communication runtime can also be used independently of the macro-based language implementation to abstract over transmission semantics, serialization and network protocols.

6.5.4 Lessons Learned

Instances of the `Transmittable` type class, which implement the transmission semantics for a specific type, can be nested, e.g., transmitting an n-tuple requires a `Transmittable` instance for every of the n elements. For accessing a value of type `T` remotely, the compiler has to resolve both a `Transmittable` instance and a serializer from the implicit scope. In our experience, failing to resolve a (deeply) nested implicit value leads to less-than-optimal error messages by the compiler because the compiler only reports on the outermost implicit that could not be resolved since it is generally not clear for which of the possibly multiple alternatives for resolving nested implicit values a developer expected implicit resolution to succeed. In our use case, we expect that `Transmittable` instances should always be resolvable or the compiler should issue an error if no matching `Transmittable` instance is in scope.

For such use cases, we propose the following scheme to achieve more precise error messages: We provide a fallback implicit value for `Transmittable`, which is defined in the `Transmittable` object. Such values are resolved by the Scala compiler with lower priority in case there is no other matching implicit value in scope. This fallback value can always be resolved but resolution results in a reference to a *compile-time-only* value, i.e., carrying the `compileTimeOnly` annotation. With this scheme, implicit resolution for `Transmittable` instances always succeeds. If there is no usable `Transmittable` instance in scope, however, the resolved fallback value results in a more meaningful compiler error that hints at the nested `Transmittable` which could not be resolved.

Another issue with Scala implicits we encountered is their lack of global coherence guarantees. In contrast to Haskell, where type class instances are globally unique, i.e., there is at most one type class implementation for every type in the whole program, Scala allows different type class instances for the same type. Prece-

dence rules for implicit resolution decide which type class instance is chosen by the compiler. Coherence is important for our use case, since the definition site of a placed value – and the generated dispatch logic for remote accesses (Section 6.4.2 on page 112) – and the remote call site of a placed value (Section 6.4.3 on page 115) need to agree on the transmission semantics implemented by the `Transmittable` type class instance. By inspecting the AST, containing the values implicitly resolved by the compiler, during macro expansion, we ensure that `Transmittable` instances are coherent or issue a compiler error otherwise.

7 A Design and Performance Evaluation

> *Genuine research takes time … sometimes a lifetime of painstaking, detailed work in order to get any results.*
>
> **— Beverly Crusher**

This chapter evaluates ScalaLoci with open-source case studies and side-by-side comparisons of alternative designs of applications belonging to different domains (e.g., big data processing, real-time streaming, games, collaborative editing, instant messaging), covering both the compilation of Scala to Java bytecode and to JavaScript via Scala.js. We conduct performance benchmarks applying ScalaLoci in a real-world setting and microbenchmarks to isolate the performance impact of the provided abstractions.

7.1 Research Questions

The main hypothesis of ScalaLoci's design is that its multitier reactive abstractions reduce the complexity of implementing a distributed system at negligible cost and that multitier modules enable the separation of modularization and distribution. The objective of the evaluation is to assess the design goals established in Chapters 3 and 4 on page 53 and on page 75, answering the following research questions:

RQ1 Does ScalaLoci improve the design of distributed applications?

RQ2 Do ScalaLoci multitier modules enable defining reusable patterns of interaction between distributed components?

RQ3 Do ScalaLoci multitier modules enable separating the modularization and distribution concerns?

RQ4 What is the performance impact introduced by ScalaLoci?

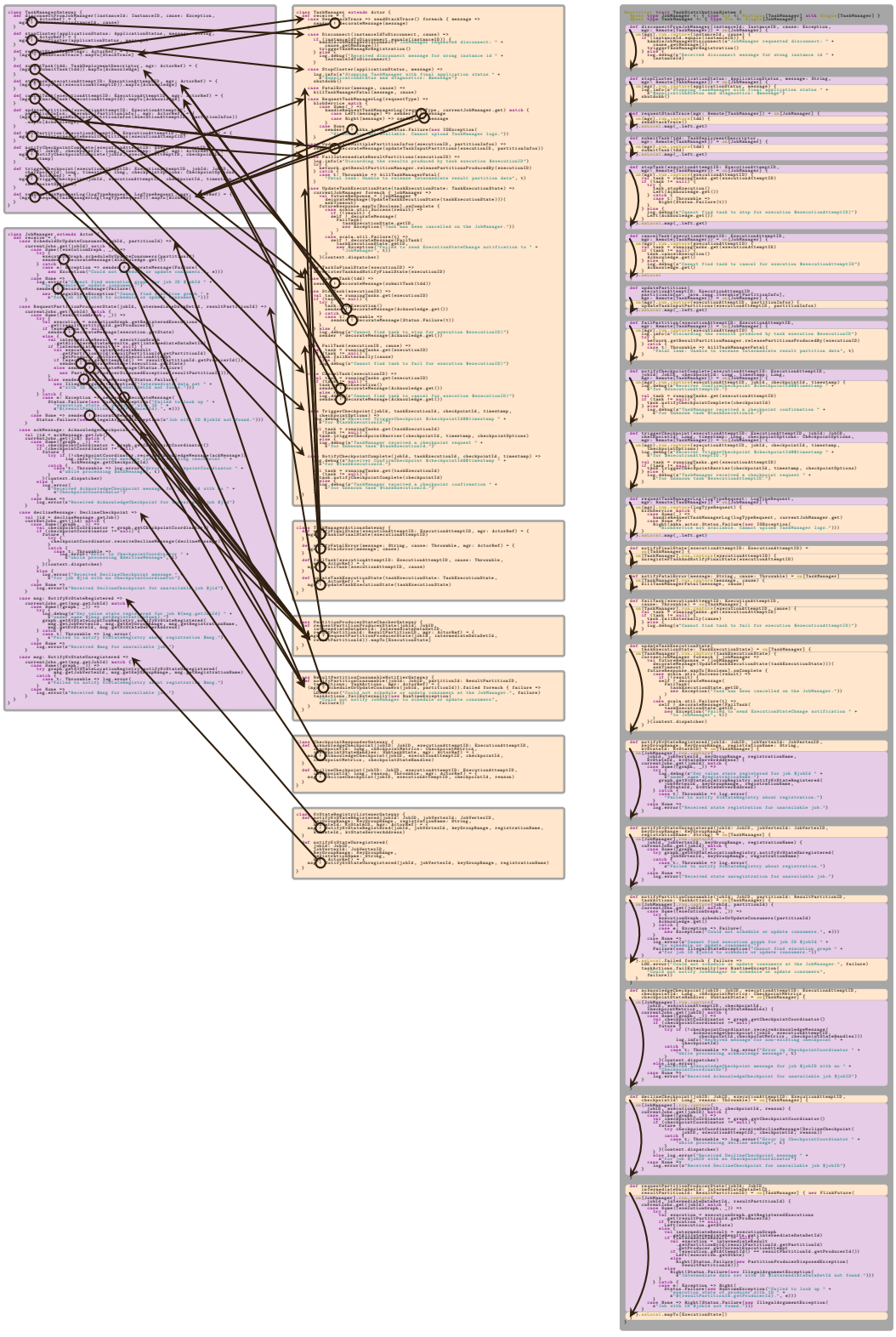
7.2 Case Studies

To evaluate the applicability of ScalaLoci to existing real-word software, we ported several open source applications. Our ports are *not* simplified versions. We reimplemented components of the existing software in ScalaLoci to achieve a functionally equivalent system, replacing the communication between certain components with a ScalaLoci implementation and keeping the the original implementation for other functionality.

7.2.1 Apache Flink

We reimplemented the *task distribution system* of the Apache Flink stream processing framework [Carbone et al. 2015] in ScalaLoci, which provides Flink's core task scheduling and deployment logic. It consists of the coordinator of the Flink instance, the *JobManager* and one or more *TaskManagers*, which execute tasks – parts of the job – deployed by the JobManager. The task distribution system is based on Akka actors and consists of 23 remote procedures in six *gateways* – an API that encapsulates sending actor messages into asynchronous RPCs – amounting to $\sim$500 SLOC of highly complex Scala code. Every gateway needs to explicitly handle each message in the actor message loop and send a response message. Invoking the message sending operation of the actor framework is asynchronous and returns immediately. Hence, it's not obvious for a developer where a message will be handled without inspecting larger parts of the code base. 19 out of the 23 RPCs are processed in a different compilation unit within another package, impeding to correlate sent messages with the remote computations they trigger. The ScalaLoci version replaces the sending and receiving operations between actors – used in the gateway implementation to implement RPCs – with remote blocks as high-level communication abstraction. Cross-peer control and data flow is explicit, thus much easier to track.

Figure 7.1 on the next page provides an overview of the communication between the JobManager and a TaskManager. Figure 7.1a on the facing page shows the code of the TaskManagerGateway used by the JobManager actor (dark violet boxes on the left) and its communication (arrows) with the TaskManager actor (light orange boxes on the right). Figure 7.1b on the next page shows the ScalaLoci implementation (deep gray box in the background) of the JobManager peer (dark violet) and the TaskManager peer (light orange). Flink's actor-based approach intertwines data flow between components with send and receive operations. Overall, the data flow is hard to track (Figure 7.1a on the facing page). Yet, data flow between both components is quite *regular*, with the JobManager triggering remote computations



(a) Original Flink.

(b) ScalaLoci Flink.

Figure 7.1 Communication for two actors in Flink.

on the TaskManager and the TaskManager – in some cases – returning a result. This *regularity* is directly captured using ScalaLoci's remote blocks (Figure 7.1b on the previous page).

Flink communication is unsafe, with actor messages having type `Any` requiring downcasting or non-exhaustive pattern matching. Crucially, the compiler cannot enforce that a message is even handled to produce a remote result. In the ScalaLoci version, we were able to eliminate 23 unsafe pattern matches and 8 type casts.

An example instance of the aspects above – simpler control flow and increased type safety – is shown in Section 7.4.3, Listing 7.5 on page 146, which provides a side-by-side comparison of a simplified code excerpt of the communication between the the JobManager and the TaskManager.

Summary The case study demonstrates how cross-host communication can be structured in more direct way using remote blocks to delimit the parts of the task distribution functionality that should be executed on the remote TaskManager explicitly. The ScalaLoci module encapsulates the complete task distribution system, consolidating the local and remote parts which belong to the same functionality – traditionally scattered over different modules – in the same multitier module (RQ1).

Crossing host boundaries is explicit and type-safe by using remote blocks as high-level communication abstraction.

7.2.2 Apache Gearpump

Apache Gearpump [Zhong et al. 2014] is a real-time streaming engine. In Gearpump, *Master* actors allocate processing tasks to *Worker* actors and collect results. A *MasterProxy* actor assigns Workers to Masters. We ported the assignment logic ($\sim$100 SLOC) to ScalaLoci, modeling the role of each of the three types of actors by a different peer. We replaced actor communication with multitier reactive abstractions. The MasterProxy message loop mixes largely unrelated tasks, e.g., assigning Workers to a Master and monitoring the Master for termination, which are captured by separated multitier event streams in the ScalaLoci version. We also removed imperative state changes on the Master – managing the list of currently connected Workers – and on the Worker – keeping track of the currently connected Master. In ScalaLoci, the list of connected remote instances is automatically handled by the runtime. Finally, the constraint that a Master can connect to multiple Workers and a Worker can connect to at most one Master is enforced at compile time in the ScalaLoci version with in-language architecture definitions.

To give an intuition of how our reimplementation compares to the original Gearpump implementation, Listing 7.1 on the next page provides a side-by-side

Listing 7.1 Master–worker assignment in Apache Gearpump.

(a) Original Gearpump implementation.

```
1.a  class MasterProxy extends Actor {
2.a    def findMaster() =
3.a      config.masters foreach {
4.a        _ ! Identify() }
5.a    findMaster()
6.a
7.a    def receive = establishing
8.a
9.a    def establishing: Receive = {
10.a     case ActorIdentity(master) =>
11.a       context.watch(master)
12.a       context.become(
13.a         active(master)) }
14.a
15.a   def active(master: ActorRef)
16.a       : Receive = {
17.a     case Terminated(master) =>
18.a       context.become(establishing)
19.a       findMaster()
20.a     case message @ Register =>
21.a       master forward message }
22.a }
```

```
1.b  class Worker(masterProxy: ActorRef)
2.b      extends Actor {
3.b    masterProxy ! Register
4.b    var master: ActorRef = null
5.b    def receive = {
6.b      case Registered(master) =>
7.b        this.master = master }
8.b  }
```

```
1.c  class Master extends Actor {
2.c    var workers = Set.empty[ActorRef]
3.c    def receive = {
4.c      case Register =>
5.c        workers += sender
6.c        sender ! Registered(self) }
7.c  }
```

(b) Refactored ScalaLoci implementation.

```
1  on[MasterProxy] {
2    def findMaster() =
3      config.masters foreach {
4        remote[Master].connect }
5    findMaster()
6
7    (remote[Master].connected
8     changedTo Seq.empty
9     observe { _ => findMaster() })
10 }
11
12 val register =
13   on[Worker] { Event[Path]() }
14
15 val registered = on[MasterProxy] sbj {
16     master: Remote[Master] =>
17   (register.asLocalFromAllSeq
18    collect { case (_, path)
19      if remote[Master].connected()
20         .headOption contains master =>
21        path })
22 }
23
24 on[Master] {
25   (registered.asLocalFromAllSeq
26    observe { remote[Worker].connect })
27 }
28
29 on[Worker] { register.fire(path) }
```

comparison for a heavily simplified but representative code excerpt of the original actor-based implementation of assigning Workers to a Master and the ScalaLoci reimplementation using multitier reactives.

In the original Gearpump implementation (Listing 7.1a), the `findMaster` method pings possible Masters (Line 4.a). The first Master that answers becomes the *active* Master to which Workers are assigned (Line 12.a). The MasterProxy *watches* the Master (Line 11.a) to get a notification when the Master terminates to initiate the search for a new Master, which (1) mixes the monitoring logic for Masters (Line 17.a) with the application logic of assigning Workers (Line 20.a) within a single message

loop. Workers register with a Master by sending a message to the MasterProxy (Line 3.b), which forwards the request to the active Master (Line 21.a), (2) scattering the control flow over three actors, which makes the application logic hard to understand. The Master adds every assigned Worker to its local list of Workers (Line 5.c) and notifies the Worker (Line 6.c), which sets the associated Master (Line 7.b), thereby manipulating local state explicitly for (3) manually maintaining the architecture of multiple Workers connected to a single Master.

We reimplemented the Worker assignment in ScalaLoci (Listing 7.1b on the previous page). The monitoring logic for Masters is implemented in one code block (Lines 1–10) and (1) not entangled with the application logic of assigning Workers (Lines 15–22). The MasterProxy pings possible Masters (Line 9) when the list of connected Master instances becomes empty (Line 8). Registration events from the Workers are forwarded to the active Master – which is the first Master in the list of connected Masters (Line 20) – via `registered` (Line 15). The (2) cross-peer data flow from `register` over `registered` to the `observe` method is explicit through `register.asLocalFromAllSeq` (Line 17) and `registered.asLocalFromAllSeq` (Line 25). Thanks to ScalaLoci's in-language architecture definition, it is not necessary to (3) mingle the architectural model with the application logic by manually maintaining a list of connected Workers on the Master and the currently connected Master on the Workers. The connected remote instances are available through ScalaLoci's `remote[Worker].connected` and `remote[Master].connected` signals, respectively. Gearpump's architecture is described by the following tie specification:

```
1 @peer type MasterProxy <: { type Tie <: Multiple[Master] with Multiple[Worker] }
2 @peer type Worker <: { type Tie <: Single[MasterProxy] with Optional[Master] }
3 @peer type Master <: { type Tie <: Multiple[MasterProxy] with Multiple[Worker] }
```

Summary The case study shows how ScalaLoci's declarative architecture specification relieves the developer of the need to manually manage state representing the system's current distribution topology, e.g., the Master automatically maintains a list of connected workers. Further, multitier reactives implement (cross-host) data flow in a direct style, avoiding indirection through an actor message loop (RQ1).

7.2.3 Play Scala.js Application

We ported the *Play Framework with Scala.js Showcase* application [Puripunpinyo 2014], an open source demonstrator of Scala.js combined with the Play Framework [Lightbend 2007], to ScalaLoci. It implements several components amounting to ∼1 500 SLOC, including instant messaging and a reactive Todo list [Li 2013]

based on TodoMVC [TodoMVC Development Team 2011]. The ScalaLoci version is feature-equivalent to the original one except for the Todo list where the updates made by a client are automatically propagated to all other clients using ScalaLoci's multitier reactives. In contrast, the original version requires to reload the page to propagate changes. We were able to reuse 70 % SLOC just by combining highly coupled code for handling client–server interaction into a multitier module and adding placement annotations. Send and receive operations for transferring values are implemented inside different modules in the original baseline. Communication on the server side is handled by a *controller*, which defines *actions*, i.e., a callback mechanism to handle HTTP client requests and create a response or setup communication channels over WebSocket. Clients send requests using Ajax and handle responses using callbacks. With ScalaLoci, all communication code is automatically generated. The communication boilerplate – sending messages with the web browser WebSocket API and callbacks, Play's *iteratees* mechanism for the receiving channel and *enumerators* for sending channel, or message serialization to JSON – is reduced by 63 %. The only boilerplate code left is needed for integration with the *Play* framework.

Summary In the case study, multitier reactives simplify cross-host data flow, enhancing the functionality of the TodoMVC component with automatic updates propagating to all connected clients. The case study further suggests that ScalaLoci reduces communication boilerplate and enables abstraction over the differences of the underlying execution platform, enabling a design that is more focused on the actual application logic (RQ1).

7.3 Variants Analysis

To evaluate the design of the applications that use ScalaLoci, we compare different variants of the same software. We reimplemented every variant from scratch to provide a fair comparison when exchanging different aspects of the implementation, i.e., the communication mechanism and the event processing strategy. For each of the 22 variants (each line in Table 7.1 on the following page), we report both aspects. For example, *Akka / observer* adopts Akka actors for the communication and no reactive features for local event propagation. The *local* variant is a non-distributed, purely local baseline. The *ScalaLoci* variants use multitier abstractions. All other variants use manually written communication code between distributed components, i.e., Akka actors, Java Remote Method Invocation (RMI) for JVM applications or native web browser APIs (WebSocket or WebRTC) for JS applications.

Table 7.1 Code metrics.

Case Study Communication / Computation	Lines of code	Callbacks	Imperative State Updates	Cross-Host Composition	Remote Access, Reads, Writes
Pong					
(local) / observer	355	17	10	0	0
(local) / reactive	327	8	0	0	0
RMI / observer	460	24	14	0	5
RMI / reactive	431	13	6	0	5
Akka / observer	440	24	13	0	5
Akka / reactive	413	18	5	0	5
ScalaLoci / observer	426	22	13	0	7
ScalaLoci / reactive	369	8	0	4	4
Shapes					
WebSocket / observer (JS) [a]	483	11	10	0	9
WebSocket / observer [b]	474	11	10	0	7
WebSocket / reactive [b]	462	9	5	0	3
Akka / observer [b]	478	13	9	0	7
Akka / reactive [b]	424	11	4	0	3
ScalaLoci / observer [b]	350	9	7	2	9
ScalaLoci / reactive [b]	345	2	0	6	6
P2P Chat					
WebRTC / observer (JS) [a]	776	22	25	0	7
WebRTC / observer [b]	824	24	25	0	7
WebRTC / reactive [b]	820	14	10	0	7
Akka / observer [b]	772	25	24	0	7
Akka / reactive [b]	771	15	9	0	7
ScalaLoci / observer [b]	637	21	19	4	8
ScalaLoci / reactive [b]	593	4	4	7	7

[a] Uses handwritten JavaScript for the client-side, Scala for the server side.
[b] All code is in Scala or ScalaLoci. The client is compiled to JavaScript via Scala.js.

The variants marked with *JS* use handcrafted JavaScript for the browser side. The other variants are compiled from Scala.

Pong implements the arcade *Pong* game where two players hit a ball with one racket each to keep it on the screen. We extended a local implementation (the user plays against the computer) to a distributed multiplayer implementation in which two users play against each other The latter adopts a client–server model. Both the server and the clients run on the JVM. *Shapes* is a collaborative drawing web application for basic geometric shapes, where clients connect to a central server. *P2P Chat* is the P2P web chat application introduced in the initial running example of Chapter 3 on page 53, which supports multiple one-to-one chat sessions. Peers communicate directly in a P2P fashion after discovery via a registry. In the latter two cases, the server and the registry run on the JVM whereas clients and peers run in the web browser.

Programming experience To give an intuition of the experience of using ScalaLoci, Figure 7.2 on the facing page shows a side-by-side comparison of the four different reactive *Pong* variants. We exclude the GUI from the code excerpts and from the

(a) Local. **(b)** ScalaLoci. **(c)** Akka. **(d)** RMI.

Figure 7.2 Pong variants.

following discussion as it is the same for all variants. We highlight the additional code needed for the distributed versions in Figures 7.2b to 7.2d compared to the local baseline in Figure 7.2a. We use yellow for code added to enable the multiplayer game. Orange indicates code added for distribution.

Transforming the local variant of *Pong* into a distributed application is straightforward. In the ScalaLoci variant (Figure 7.2b), the majority of the additions are 8 SLOC which implement the multiplayer functionality. As much as 91 % SLOC (excluding the GUI) could be reused from the local version just by adding placement to assign values to either the client or the server and adding remote accesses via `asLocal`. The reused code amounts to 72 % SLOC of the ScalaLoci version. In the Akka and RMI versions (Figures 7.2c and 7.2d), we could also reuse 91 % SLOC of the local baseline, which is 53 % SLOC of the Akka version and 46 % SLOC for RMI. The explanation is that the Akka and RMI versions lack multitier reactives, thus data flows cannot be directly defined across client and server boundaries. Instead, message passing and callbacks or imperative remote calls are needed to explicitly propagate changes across hosts – which constitutes ∼ 60 % of the added code (orange) – tangling the application logic with connection management and data propagation.

Code metrics and safety Table 7.1 on the preceding page compares code metrics for all variants (rows). Not surprisingly, ScalaLoci requires less code to express the same functionality as compared to a non-multitier approach. Multitier reac-

tives further reduce the code size and the amount of callbacks (*Callbacks* column). We use the number of callbacks as a measure of design quality, since replacing callbacks – which are not composable – by composable reactives removes manual imperative state updates (*Imperative State Updates* column) and improves extensibility [Meyerovich et al. 2009]. Reactives exhibit better composability properties compared to callbacks due to the inversion of control problem [Maier et al. 2010]. To completely eliminate callbacks in reactive programming, also the libraries used in the application code need to support reactive abstractions, e.g., a text input widget in a UI library need to expose a signal holding the current text. Since most libraries adopt callbacks at their boundaries, some callbacks are still necessary even if the application logic and cross-host communication is based on reactive programming.

In ScalaLoci, data flows on different hosts can be seamlessly composed (*Cross-host Composition* column), which is not possible in approaches where data flow across hosts is interrupted, e.g., by RPC or message sending. The applications introduced in Section 3.3 on page 63 compose data flows across hosts. For instance, the token ring example (Listing 3.5 on page 67, Line 16) composes the local events `recv` and `sendToken` and the remote event `sent` seamlessly inside the single expression `(sent.asLocal \ recv) || sendToken`.

Statically typed access to remote values in ScalaLoci ensures that they are serializable, opposed to Java RMI serialization, which can fail at run time. Such remote accesses (*Remote Access* column) involve potentially problematic serialization in non-ScalaLoci cases. For example, the variants using manually written communication code explicitly invoke methods to convert between in-memory and serialized representation, e.g., `JSON.stringify(`v`)` and `JSON.parse(`v`)` in JavaScript code or `write(`v`)` and `read[`T`](`v`)` in Scala code. Also, remote access is achieved with explicit sending and receiving operations. Since the compiler cannot enforce that the types on both sides match, it cannot prevent that values are manipulated inconsistently, resulting in run time errors. In ScalaLoci, instead, a shared value is accessed locally as v and remotely as v`.asLocal`. Hence the compiler can statically check that v's type is consistent for local and remote accesses.

Summary The comparison of the different variants suggests that there exist quantifiable differences in design regarding the number of used imperative callbacks compared to reactive abstractions and regarding abstractions for composing local and remote values, which we interpret as ScalaLoci enabling improvement in software design, fostering more concise and composable code and reducing the number of callbacks and imperative state updates (RQ1). Also, applications in ScalaLoci are safer than their counterparts, e.g., due to reduced risk of run time type mismatches thanks to static type-checking across peers.

7.4 Modularity Studies

To evaluate the ScalaLoci module system, we first consider *distributed algorithms* as a case study. Distributed algorithms are a suitable case study because – as we describe in the next section – they depend on each other and on the underlying architecture. We show how to keep each algorithm modularized in a way that algorithms can be freely composed.

Second, we demonstrate how *distributed data structures* can be implemented in ScalaLoci. This case study requires to hide the internal behavior of the data structure from user code as well as to provide a design that does not depend on the specific system architecture. We further evaluate the applicability of the ScalaLoci module system to existing real-word software, building on the ScalaLoci reimplementation of the Apache Flink distributed stream processing framework's *task distribution system* introduced in Section 7.2.1 on page 126.

7.4.1 Distributed Algorithms

We present a case study on a distributed algorithm for mutual exclusion through global locking to access a shared resource. As global locking requires a leader election algorithm, we implement different election algorithms as reusable multitier modules. Also, leader election algorithms assume different distributed architectures, which we represent as multitier modules, too.

The implemented mechanism relies on a central coordinator (Listing 7.2 on the next page). The `MutualExclusion` module is a constrained multitier module (Section 4.3 on page 85) that is parameterized over the leader election algorithm by specifying a requirement on the `LeaderElection` interface (Line 2), abstracting over concrete leader election implementations. `LeaderElection` provides the `state` value (Lines 7 and 15) indicating whether the local node is the elected leader. The `MutualExclusion` module defines the `lock` (Line 6) and the `unlock` (Line 14) methods to acquire and release the lock. Calling `lock` returns `true` if the lock was acquired successfully, i.e., the lock was not already acquired yet. Calling `unlock` releases the lock. Only the elected leader can grant locks to peer instances.

System architectures The `MutualExclusion` module (Listing 7.2 on the next page) specifies a constraint on `Architecture` (Line 2) requiring any distributed architecture for the system abstracting over a concrete one. `Architecture` is the base trait for different distributed architectures expressed as reusable modules. This way, module definitions can abstract over the concrete distributed architecture, and module implementations can be instantiated to different architectures (as shown

Listing 7.2 Mutual exclusion.

```
@multitier trait MutualExclusion[T] {
    this: Architecture with LeaderElection[T] =>

  private var locked: Local[Option[T]] on Node = None

  def lock(id: T): Boolean on Node =
    if (state == Leader && locked.isEmpty) {
      locked = Some(id)
      true
    }
    else
      false

  def unlock(id: Id): Unit on Node =
    if (state == Leader && locked.contains(id))
      locked = None
}
```

for the `MasterWorker` module in Section 4.4 on page 87). Each peer type defined in an architecture extends `Node` allowing common functionalities to be placed on `Node`. Listing 7.3 on the next page shows the definitions for different architectures with their iconification on the right. The `Architecture` module defines the general `Node` peer and the constraint that peers of type `Node` are connected to an arbitrary number of other `Node` peers. With no other constraints, other architecture modules can specialize `Architecture`. `Node` therefore serves as a common super type for peer definitions in other architectures.

All distributed peers specified in Listing 7.3 on the facing page are `Node` peers. The `P2P` module defines a `Peer` that can connect to arbitrary many other peers. Thus, `P2P` is essentially the general architecture since nodes connecting in a peer-to-peer fashion do not impose any additional architectural constraints. The `P2PRegistry` module adds a central registry to which peers can connect. The `MultiClientServer` module defines a client that is always connected to single server, while the server instance can handle multiple clients simultaneously. The `ClientServer` module specifies an architecture where a single server instance always handles a single client. For the `Ring` module, we define a `Prev` and a `Next` peer. A `RingNode` itself is both a predecessor and a successor. All `Node` peers have a single tie to their predecessor and a single tie to their successor.

Leader election We present the `LeaderElection` interface for a generic leader election algorithm as ScalaLoci module. Since leader election differs depending on the network architecture, the interface defines a self-type constraint on `Architecture`, abstracting over the concrete network architecture constraining

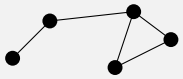
Listing 7.3 Distributed architectures.

(a) Base architecture.

```
@multitier trait Architecture {
  @peer type Node <: { type Tie <: Multiple[Node] }
}
```

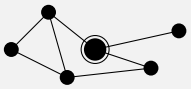
(b) Peer-to-peer architecture.

```
@multitier trait P2P extends Architecture {
  @peer type Peer <: Node { type Tie <: Multiple[Peer] }
}
```



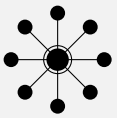
(c) Peer-to-peer with central registry architecture.

```
@multitier trait P2PRegistry extends P2P {
  @peer type Registry <: Node {
    type Tie <: Multiple[Peer] }
  @peer type Peer <: Node {
    type Tie <: Optional[Registry] with Multiple[Peer] }
}
```



(d) Multiple client and single server architecture.

```
@multitier trait MultiClientServer extends Architecture {
  @peer type Client <: Node {
    type Tie <: Single[Server] with Single[Node] }
  @peer type Server <: Node {
    type Tie <: Multiple[Client] }
}
```

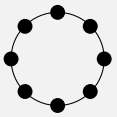


(e) Single client and single server architecture.

```
@multitier trait ClientServer extends MultiClientServer {
  @peer type Client <: Node {
    type Tie <: Single[Server] with Single[Node] }
  @peer type Server <: Node {
    type Tie <: Single[Client] }
}
```

(f) Ring architecture.

```
@multitier trait Ring extends Architecture {
  @peer type Node <: { type Tie <: Single[Prev] with Single[Next] }
  @peer type Prev <: Node
  @peer type Next <: Node
  @peer type RingNode <: Prev with Next
}
```



multitier mixing (Section 4.3 on page 85). Further, the interface abstracts over a mechanism for assigning IDs to nodes implemented by the `Id[T]` module:

```
@multitier trait LeaderElection[T] { this: Architecture with Id[T] =>
  def state: State on Node
  def electLeader(): Unit on Node
  def electedAsLeader(): Unit on Node
}
```

The type parameter `T` of the `Id` module represents the type of the IDs. The `Id` module interface defines a local `id` value on every node and requires an ordering relation for IDs:

```
@multitier abstract class Id[T: Ordering] { this: Architecture =>
  val id: Local[T] on Node
}
```

The following example implementation for the `Id` interface generates a random ID for every node:

```
@multitier trait RandomIntId extends Id[Int] { this: Architecture =>
  val id: Local[Int] on Node = new java.security.SecureRandom().nextInt()
}
```

The `LeaderElection` module defines a local variable `state` that captures the state of each peer (e.g., `Candidate`, `Leader` or `Follower`). The `electLeader` method is kept abstract to be implemented by a concrete implementation of the interface. After a peer instance has been elected to be the leader, implementations of `LeaderElection` call `electedAsLeader`. We consider three leader election algorithms:

Hirschberg-Sinclair leader election The Hirschberg-Sinclair leader election algorithm [Hirschberg and Sinclair 1980] implements leader election for a ring topology. In every algorithm phase, each peer instance sends its ID to both of its neighbors in the ring. IDs circulate and each node compares the ID with its own. The peer with the greatest ID becomes the leader. The logic of the algorithm is encapsulated into the `HirschbergSinclair` module, which extends `LeaderElection`:

```
@multitier trait HirschbergSinclair[T] extends LeaderElection[T] {
    this: Ring with Id[T] =>

  def electLeader() = on[Node] { elect(0) }
  private def elect(phase: Int) = on[Node] { /* ... */ }
  private def propagate(remoteId: T, hops: Int, direction: Direction) =
    on[Node] { /* ... */ }
}
```

The module's self-type encodes that the algorithm is designed for ring networks (Line 2). When a new leader election is initiated by calling `electLeader` (Line 4), the `elect` method is invoked (Line 5). The `propagate` method passes the IDs of peer instances along the ring and compares them with the local ID.

Yo-Yo leader election The Yo-Yo algorithm [Santoro 2006] is a universal leader election protocol, i.e., it is independent of the network architecture. Hence, the self-type of the `YoYo` implementation is simply `Architecture with Id[T]`. In the Yo-Yo algorithm, each node exchanges its ID with all neighbors, progressively pruning subgraphs where there is no lower ID. The node with the lower ID becomes the leader.

Raft leader election The Raft consensus algorithm [Ongaro and Ousterhout 2014] elects a leader by making use of randomized timeouts. Upon election, the leader maintains its leadership by sending heartbeat messages to all peer instances. If instances do not receive a heartbeat message from the current leader for a certain amount of time, they initiate a new election.

Instantiating global locking The following code instantiates a `MutualExclusion` module using the Hirschberg-Sinclair leader election algorithm for a ring architecture:

```
1 @multitier object locking extends
2   MutualExclusion[Int] with HirschbergSinclair[Int] with Ring with RandomIntId
```

The example mixes in a module implementation for every module over which other modules are parameterized, i.e., `MutualExclusion` is parameterized over `LeaderElection`, which is instantiated to the `HirschbergSinclair` implementation. `HirschbergSinclair` requires a `Ring` architecture and an `Id` implementation, which is instantiated to the `RandomIntId` module. The following code, instead, instantiates a `MutualExclusion` module using the Yo-Yo leader election algorithm for a P2P architecture:

```
1 @multitier object locking extends
2   MutualExclusion[Int] with YoYo[Int] with P2P with RandomIntId
```

Summary The case study demonstrates how module implementations for concrete architectures and leader election algorithms can be composed into a module providing global locking and can be made reusable. Since modules encapsulate a functionality within a well-defined interface, leader election algorithms can be easily exchanged. Our approach allows for simply mixing different cross-peer functionality together without changing any multitier code that is encapsulated into the modules (RQ2).

Table 7.2 Common conflict-free replicated data types.

CRDT	Description	Lines of Code		Remote Accesses
		Scala *local*	ScalaLoci *distributed*	
G-Counter	Grow-only counter. Only supports incrementing.	14	15	1
PN-Counter	Positive-negative counter. Supports incrementing and decrementing.	13	14	1
LWW-Register	Last-write-wins register. Supports reading and writing a single value.	10	11	1
MV-Register	Multi-value register. Supports writing a single value. Reading may return a set of multiple values that were written concurrently.	12	13	1
G-Set	Grow-only set. Only supports addition.	7	9	1
2P-Set	Two-phase set. Supports addition and removal. Contains a G-Set for added elements and a G-Set for removed elements (the *tombstone set*). Removed elements cannot be added again.	13	17	2
LWW-Element-Set	Last-write-wins set. Supports addition and removal. Contains a set for added elements and a set for removed elements and associates each added and removed element to a time stamp. Associates each added and removed element to a time stamp.	15	19	2
PN-Set	Positive-negative set. Supports addition and removal. Associates a counter to each element, incrementing/decrementing the counter upon addition/removal.	12	16	2
OR-Set	Observed-removed set. Supports addition and removal. Associates a set of added and of removed (unique) tags to each element. Adding inserts a new tag to the added tags. Removing moves all tags associated to an element to the set of removed tags.	15	18	2

7.4.2 Distributed Data Structures

This section demonstrates how distributed data structures can be implemented in ScalaLoci. First, we reimplement non-multitier conflict-free replicated data types (CRDTs) as multitier modules in ScalaLoci, The multitier reimplementations do not only implement conflict-free merging of distributed updates – which is already a feature of the non-multitier versions – but additionally abstract over replication among distributed components. Second, we compare to an existing multitier cache originally implemented in Eliom [Radanne and Vouillon 2018].

Conflict-free replicated data types Conflict-free replicated data types (CRDT) [Shapiro et al. 2011a,b] offer eventual consistency across replicated components for specific data structures, avoiding conflicting updates by design. With CRDTs, updates to shared data are sent asynchronously to the replicas and *eventually* affect all copies. Such *eventually consistent* model [Vogels 2009] provides better performance (no synchronization is required) and higher availability (each replica has a local copy which is ready to use). We reimplemented several CRDTs, publicly available in Scala [Li 2015], in ScalaLoci (Table 7.2 on the facing page).

We discuss the representative case of the `GSet` CRDT. G-Sets (grow-only sets) are sets which only support adding elements. Elements cannot be removed. A *merge* operation computes the union of two G-Sets. Listing 7.4a1 on the next page shows the G-Set in Scala. `GSet` defines a set `content` (Line 4) and a method to check if an element is in the set (Line 7). Adding an element inserts it into the local `content` set (Line 10). Listing 7.4b on the following page presents a multitier module for a multitier G-Set. The implementations are largely similar despite that the ScalaLoci version is distributed and the Scala version is not. The Scala CRDTs are only local. Distributed data replication has to be implemented by the developer (Listing 7.4a2 on the next page).

In the ScalaLoci variant, the peer type of the interacting nodes is abstract, hence it is valid for any distributed architecture. The ScalaLoci multitier module can be instantiated by applications for their architecture:

```
1  @multitier trait EventualConsistencyApp {
2    @peer type Client <: ints.Node with strings.Node {
3      type Tie <: Single[Server] with Single[ints.Node] with Single[strings.Node] }
4    @peer type Server <: ints.Node with strings.Node {
5      type Tie <: Single[Client] with Single[ints.Node] with Single[strings.Node] }
6
7    @multitier object ints extends GSet[Int]
8    @multitier object strings extends GSet[String]
9
10   on[Server] { ints.add(42) }
11   on[Client] { strings.add("forty-two") }
12 }
```

The example defines a `GSet[Int]` (Line 7) and a `GSet[String]` (Line 8) instance. The `Client` peer and the `Server` peers are also `ints.Node` and `strings.Node` peers (Lines 2 and 4) and are tied to other `ints.Node` and `strings.Node` peers (Lines 3 and 5). Thus, both the server (Line 10) and the client (Line 11) can use the multitier module references (Section 4.2.1 on page 80) `ints` and `strings` to add elements to both sets, which (eventually) updates the sets on the connected nodes. The plain Scala version, in contrast, does not offer abstraction over placement.

Listing 7.4 Conflict-free replicated grow-only set.

(a) Scala implementation.

(a1) Traditional G-Set implementation.

```
1  class GSet[T] {
2
3
4    val content =
5      mutable.Set.empty[T]
6
7    def contains(v: T) =
8      content.contains(v)
9
10   def add(v: T) =
11     content += v
12
13
14
15   def merge(other: GSet[T]) =
16     content ++= other.content
17 }
```

(a2) Example of user code for distribution.

```
1  trait Host[T] {
2    val set = new GSet[T]
3
4    def add(v: T) = {
5      set.add(v)
6      send(set.content) }
7
8    def receive(content: T) =
9      set.merge(content)
10 }
```

(b) ScalaLoci implementation.

```
1  @multitier trait GSet[T]
2      extends Architecture {
3
4    val content = on[Node] {
5      mutable.Set.empty[T] }
6
7    def contains(v: T) = on[Node] {
8      content.contains(v) }
9
10   def add(v: T) = on[Node] {
11     content += v
12     remote call merge(content.toSet) }
13
14   private
15   def merge(content: Set[T]) = on[Node] {
16     this.content ++= content }
17 }
```

In addition to be more concise, the ScalaLoci version exhibits a better design thanks to the combination of multitier programming and modules. In plain Scala, the actual replication of added elements by propagating them to remote nodes is mingled with the user code: The Scala versions of all CRDTs transfer updated values explicitly to merge them on the replicas, i.e., `merge` needs to be public to user code. The *Remote Accesses* column in Table 7.2 on page 140 counts the methods on the different CRDTs that mix replication logic and user code.

Listing 7.4a2 shows the user code adding an element to the local G-Set (Line 5), sending the content to a remote host (Line 6), receiving the content remotely (Line 8) and merging it into the remote G-Set (Line 9). In contrast, adding an element to the ScalaLoci `GSet` (Listing 7.4b) directly merges the updated set into all connected remote nodes (Lines 12 and 16). The multitier module implicitly performs remote communication between different peers (Line 12), encapsulating the remote access to the replicas, i.e., `merge` is private.

Distributed caching We implement a cache that is shared between a client–server architecture. It is modeled after Eliom's multitier cache holding the values already computed on a server [Radanne and Vouillon 2018]. In their example, both the clients and the server have a map of cached comments. Whenever a client connects to the server, it retrieves the map to synchronize its view on the comments. Both client and server maintain a map, which can be locally searched and filtered without remote communication. The following code presents the cache using ScalaLoci multitier modules:

```
1 @multitier trait Cache[K, V] extends MultiClientServer {
2   private val table = on[Node] { mutable.Map.empty[K, V] }
3
4   on[Client] {
5     table.asLocal foreach { serverTable => table ++= serverTable }
6   }
7
8   def add(key: K, value: V) = on[Node] { table += key -> value }
9 }
```

The `Cache` module is implemented for a client–server architecture (Line 1). The `table` map (Line 2) is placed on every `Node`, i.e., on the client and the server peer. The `add` method adds an entry to the map (Line 8). As soon as the client instance starts, the client populates its local map with the content of the server's map (Line 5).

ScalaLoci's multitier model is more expressive than Eliom's as it allows the definition of arbitrary peers through placement types. Placement types enable abstraction over placement, as opposed to Eliom, which only supports two fixed predefined places (server and client). ScalaLoci supports Eliom's client–server model (Line 1) as a special case. Thanks to ScalaLoci's abstract peer types, the `Cache` module can also be used for other architectures. For example, we can enhance the `Peer` and `Registry` peers of a P2P architecture with the roles of the client and the server of the `Cache` module by mixing `Cache` and `P2PRegistry` (Section 4.2.2 on page 81) and composing the architectures of both modules:

```
1 @multitier trait P2PCache[K, V] extends Cache[K, V] with P2PRegistry {
2   @peer type Registry <: Server {
3     type Tie <: Multiple[Peer] with Multiple[Client] }
4   @peer type Peer <: Client {
5     type Tie <: Single[Registry] with Single[Server] with Multiple[Peer] }
6 }
```

Summary The case studies demonstrate that, thanks to the multitier module system, distributed data structures can be expressed as reusable modules that can be instantiated for different architectures, encapsulating all functionalities needed for the implementation of the data structure (RQ2).

7.4.3 Apache Flink

The *task distribution system* of the Apache Flink stream processing framework [Carbone et al. 2015], introduced in Section 7.2.1 on page 126, provides Flink's core task scheduling and deployment logic. It is based on Akka actors and consists of six *gateways* (an API that encapsulates sending and receiving actor messages). Gateways wrap method arguments into messages, sending the message and (potentially) receiving a different message carrying a result.

With the current Flink design, however, code fragments that are executed on different distributed components (i.e., for sending and receiving a message), inevitably belong to different actors. The functionalities that conceptually belong to a single gateway are scattered over multiple files in the Flink implementation, breaking modularization. The messages sent by the actors in every gateway are hard to follow for developers because matching sending and receiving operations are completely separated in the code. 19 out of the 23 sent messages are processed in a different compilation unit within another package, hindering the correlation of messages with the remote computations they trigger.

We reimplemented the task distribution system using multitier modules to cover the complete cross-peer functionalities that belong to each gateway. In the ScalaLoci version, every gateway is a multitier module. The resulting modules are (1) the `TaskManagerGateway` to control task execution, (2) the `TaskManagerActions` to notify of task state changes, (3) the `CheckpointResponder` to acknowledge checkpoints, (4) the `KvStateRegistryListener` to notify key-value store changes, (5) the `PartitionProducerStateChecker` to check of the state of producers and of result partitions and (6) the `ResultPartitionConsumableNotifier` to notify of available partitions. Since the different cross-peer functionalities of the task distribution system are cleanly separated into different modules, the complete `TaskDistributionSystem` application is simply the composition of the modules 1–6 which implement each subsystem:

```
1 @multitier trait TaskDistributionSystem extends
2   CheckpointResponder with
3   KvStateRegistryListener with
4   PartitionProducerStateChecker with
5   ResultPartitionConsumableNotifier with
6   TaskManagerGateway with
7   TaskManagerActions
```

We use multitier mixing (Section 4.3 on page 85) to mix the system modules together (Lines 2–7). It is not necessary to specify the architecture of the complete task distribution system in the `TaskDistributionSystem` module since the architecture is derived from the composed subsystems, i.e., it suffices to specify the architecture

Figure 7.3 Example communication in Flink using ScalaLoci multitier modules.

in the mixed-in modules. Yet, it is of course possible to explicitly specify the task distribution system architecture as:

```
@peer type JobManager <: {
  type Tie <: Multiple[TaskManager] }
@peer type TaskManager <: {
  type Tie <: Single[JobManager] with Single[TaskManager] }
```

Compared to Figure 7.1 on page 127, which merges the functionalities of all subsystems into a single compilation unit, the ScalaLoci version using multitier modules encapsulates each functionality into a separate module. Figure 7.3 shows the `TaskDistributionSystem` module (the box in the background), composed by mixing together the subsystem modules (the boxes in the foreground). The multitier modules (the six deep gray boxes) contain code for the `JobManager` peer (dark violet) and the `TaskManager` peer (light orange). Arrows represent cross-peer data flow, which is encapsulated within modules and is not split over different modules. Importantly, even modules that place all computations on the same peer (e.g., the module containing only light orange boxes) define remote accesses (arrows), i.e., different instances of the same peer type (e.g., the light orange peer) communicate with each other. The subsystem modules contain different functionalities and abstract over their distribution – i.e., parts of the same functionality are executed on the `JobManager` peer and parts are executed on the `TaskManager` peer –

Listing 7.5 Remote communication in Flink.

(a) Original Flink implementation.

(a1) Message definition.

```
1  package flink.runtime
2
3  case class SubmitTask(
4    td: TaskDeployment)
```

(a2) Calling side.

```
1  package flink.runtime.job
2
3  class TaskManagerGateway {
4    def submitTask(
5        td: TaskDeployment,
6        mgr: ActorRef) =
7      (mgr ? SubmitTask(td))
8       .mapTo[Acknowledge]
9  }
```

(a3) Responding side.

```
1  package flink.runtime.task
2
3  class TaskManager extends Actor {
4    // standard Akka message loop
5    def receive = {
6      case SubmitTask(td) =>
7        val task = new Task(td)
8        task.start()
9        sender ! Acknowledge()
10   }
11 }
```

(b) Refactored ScalaLoci implementation.

```
1  package flink.runtime.multitier
2
3  @multitier object TaskManagerGateway {
4    @peer type JobManager <: {
5      type Tie <: Multiple[TaskManager] }
6    @peer type TaskManager <: {
7      type Tie <: Single[JobManager] }
8
9    def submitTask(
10       td: TaskDeployment,
11       tm: Remote[TaskManager]) =
12     on[JobManager] {
13       on(tm).run.capture(td) {
14         val task = new Task(td)
15         task.start()
16         Acknowledge()
17       }.asLocal
18     }
19 }
```

which allows for multitier development of distributed systems without sacrificing modularization.

It is instructive to look into the details of one of the modules. Listing 7.5 shows an excerpt of the – extensively simplified – `TaskManagerGateway` functionality for Flink (left) and its reimplementation in ScalaLoci (right) side-by-side, focusing only on a single remote access of a single gateway. The example concerns the communication between the `TaskManagerGateway` used by the `JobManager` and the `TaskManager` – specifically, the job manager's submission of tasks to task managers. In the actor-based version (Listing 7.5a), this functionality is scattered over different modules hindering correlating sent messages (Listing 7.5a2, Line 7) with the remote computations they trigger (Listing 7.5a3, Lines 7–9) by pattern-matching on the received message (Listing 7.5a3, Line 6). The ScalaLoci version (Listing 7.5b) uses a remote block (Line 13), explicitly stating the code block for the remote computation (Lines 14–16). Hence, in ScalaLoci, there is no splitting over different actors as in the Flink version, thus keeping related functionalities inside the same module. The

`TaskManagerGateway` multitier module contains a functionality that is executed on both the `JobManager` and the `TaskManager` peer. Further, the message loop of the `TaskManager` actor of Flink (Listing 7.5a3 on the preceding page), does not only handle the messages belonging to the `TaskManagerGateway` (shown in the code excerpt). The loop also needs to handle messages belonging to the other gateways – which execute parts of their functionality on the `TaskManager` – since modularization is imposed by the remote communication boundaries of an actor.

Summary In summary, in the case study, the multitier module system enables decoupling of modularization and distribution as ScalaLoci multitier modules capture cross-network functionalities expressed by Flink gateways without being constrained to modularization along network boundaries (RQ3).

7.5 Performance

We evaluate ScalaLoci performance in two ways. First, we consider the impact of ScalaLoci on a real-world system running in the cloud compared to the original implementation with Akka actors. Second, we compare communication and change propagation cost in ScalaLoci among common alternatives.

System benchmarks We measure the performance of Apache Flink compared to the ScalaLoci reimplementation presented in Sections 7.2.1 and 7.4.3 on page 126 and on page 144. Both systems are functionally equivalent and only differ in the language abstractions covered by ScalaLoci (i.e., architecture definition, placement and remote communication) used in the reimplementation of Flink's core task scheduling and deployment logic. We use the Yahoo Streaming Benchmark [Chintapalli et al. 2016] on the Amazon EC2 Cloud (2,3 GHz Intel Xeon E5-2686, 8 GiB instances) with eight servers for data generation, one server as sink, one Apache Zookeeper server for orchestration, one JobManager master server, and four to eight TaskManager worker nodes. Events are marked with timestamps at generation time. The query groups events into 10 s windows. Each data source generates 20 K events/s. The Yahoo Streaming Benchmark measures latency.

Figure 7.4 on the following page shows latency average and variance between the two versions. Figure 7.5 on the next page shows the empirical cumulative distribution: The latency for completing the processing of a given fraction of events. We consider the difference between the two versions negligible (in some cases, ScalaLoci is even faster) and only due to variability in system measurements.

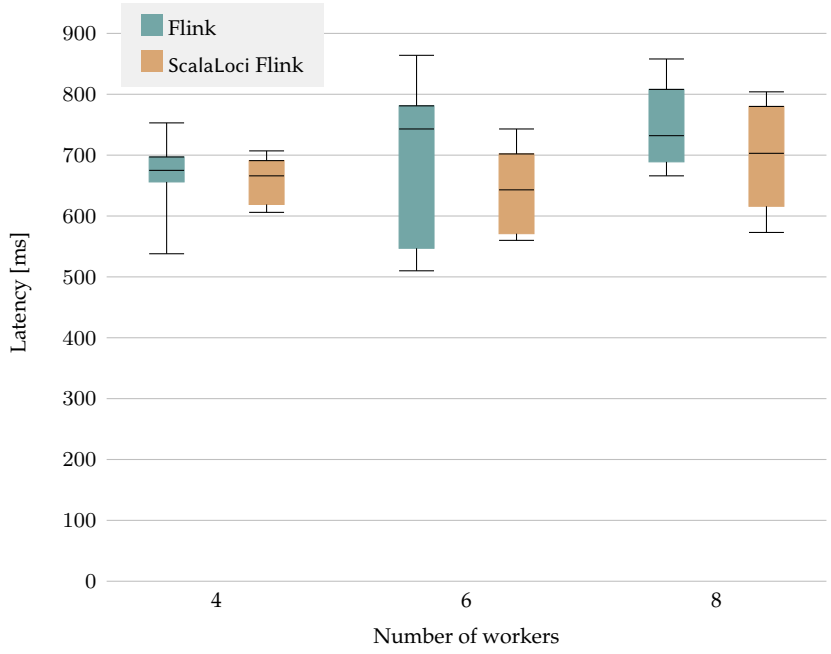


Figure 7.4 System benchmarks: Latency.

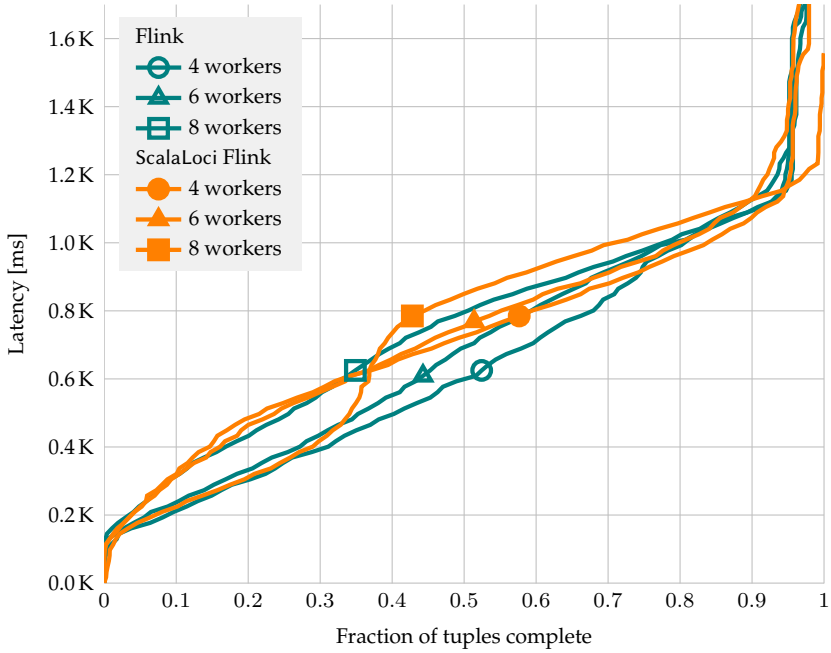


Figure 7.5 System benchmarks: Cumulative distribution.

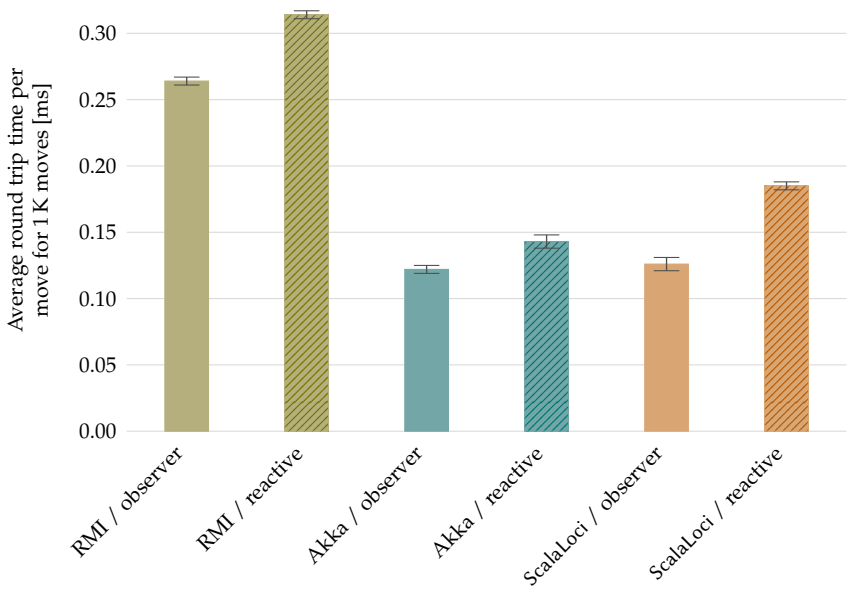


Figure 7.6 Microbenchmarks: Pong.

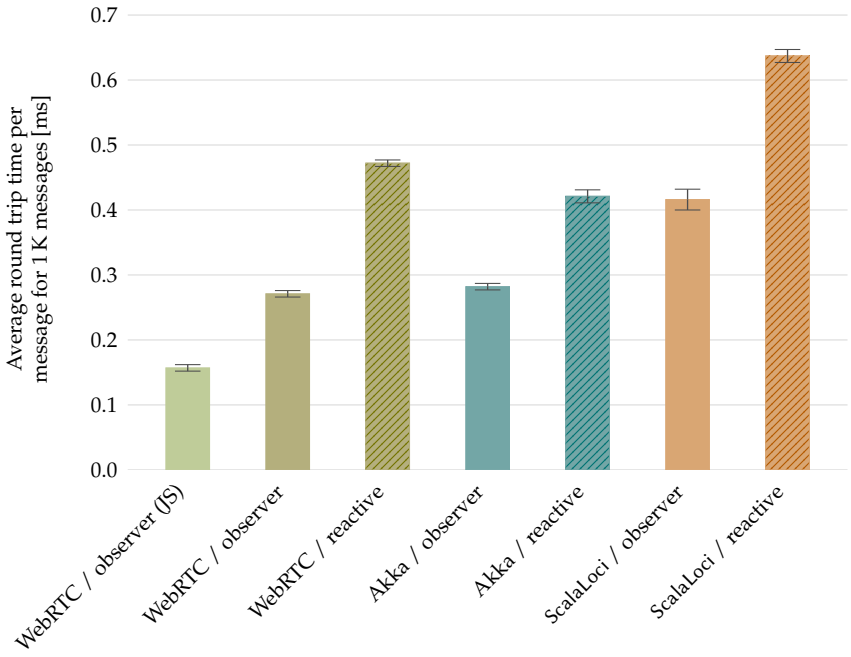


Figure 7.7 Microbenchmarks: P2P Chat.

Microbenchmarks Our microbenchmarks are based on the *Pong* and *P2P Chat* applications from Section 7.3 on page 131. *Pong* runs on the Java Virtual Machine (JVM) and *P2P Chat* runs on a JavaScript engine in a web browser. In both settings, we microbenchmark the performance impact of ScalaLoci's abstractions. We do not microbenchmark different virtual machines (i.e., running the same application on a JavaScript and a Java virtual machine) because the differences would be due to the execution environment rather than to our implementation.

To specifically only measure ScalaLoci's overhead, all peers run on the same machine. Remote communication is piped through the network stack locally and its latency is negligible. The setup is an Intel Core i7-5600U, 2.6–3.2 GHz, 64-Bit OpenJDK 8u144, 2 GB heap, Scala 2.11.8 and Scala.js 0.6.13, Debian GNU/Linux 9.2, Chromium 64-Bit v60.

The benchmarks initiate a value change which propagates to a remote peer where it triggers a computation. The result is propagated back to the initiating peer. We measure the average time over 1 K round trips. For *Pong*, we measure the time between a simulated player moves the mouse and the corresponding racket moves. For *Chat*, we measure the time between sending a chat message and receiving an answer from a chat partner. Figures 7.6 and 7.7 on the previous page show the mean over 200 iterations and the 99 % confidence intervals.

For *Pong*, ScalaLoci even outperforms the RMI variants (Figure 7.6 on the preceding page). We attribute this result to RMI's blocking remote call approach causing the application to block upon sending a changed value before propagating the next change. The performance of the ScalaLoci variants is comparable to the Akka variants. Using our reactive runtime system incurs a small overhead of ~ 0.02–0.05 ms as compared to handcrafted value propagation for both the RMI and ScalaLoci variants.

For the *P2P Chat* benchmark (Figure 7.7 on the previous page), the observer variant with handwritten JavaScript for the client is the fastest. Using higher level of abstractions adds a layer of indirection with a performance cost. Akka, ScalaLoci, and the compilation to JavaScript show an overhead partially due to the Scala.js translation. Akka.js on the client-side has a comparable performance to plain Scala.js. This is mostly because remote messages in Akka.js are sent using the same WebRTC browser API as in the *WebRTC* cases, resulting in the same amount of overhead. The overhead amounts to ~ 0.1 ms for using Scala.js to compile Scala to JavaScript (bar labeled *WebRTC / observer (JS)* compared to *WebRTC / observer* in the graph), ~ 0.15 ms for ScalaLoci compiled to JavaScript (bars labeled *ScalaLoci* compared to *WebRTC* in the graph) and ~ 0.15–0.35 ms for the reactive runtime compiled to JavaScript (bars labeled *reactive* compared to *observer* in the graph).

Summary The benchmarks show that, at the system level, there is no observable performance penalty when using ScalaLoci. Further, all microbenchmark measurements are under 1 ms – distinctly lower than network latency for Internet applications, which is in line with the results of the system benchmarks suggesting that the overhead of ScalaLoci is negligible for distributed applications (RQ4).

8 Conclusion

> *What we leave behind is not as important as how we've lived.*
>
> **— Jean-Luc Picard**

In this dissertation, we introduced the ScalaLoci distributed programming language, our approach to tackle the complexity of distributed application development through dedicated programming language support. We first summarize the contributions of this thesis and then outline opportunities for future research.

8.1 Summary

We presented ScalaLoci, a multitier reactive language with statically typed specification of data flows, spanning over multiple hosts. ScalaLoci provides language abstractions to define the architectural scheme for the distributed application, specify value distribution to different hosts via explicit placement and seamlessly compose distributed reactives. We provided a formal model for placement types and proved it sound.

Our multitier module system allows developers to modularize multitier code, enabling encapsulation and code reuse. Thanks to abstract peer types, multitier modules capture patterns of interaction among components in the system, enabling their composition and the definition of module constraints.

We further presented the ScalaLoci implementation and its design as a domain-specific language embedded into Scala. We have shown how ScalaLoci exploits Scala's advanced language features, i.e., type level and macro programming, for embedding ScalaLoci abstractions and reported on our experiences with such an approach and the challenges we needed to tackle and how they led to the current ScalaLoci architecture and implementation strategy.

The evaluation on case studies, such as distributed algorithms and data structures, and third party systems, such as the Apache Flink task distribution system and the Apache Gearpump real-time streaming engine, shows higher design quality at the cost of negligible performance overhead. The evaluation further demonstrates that

ScalaLoci's multitier module system is effective in properly modularizing distributed applications.

8.2 Perspectives

In perspective, this thesis opens up several directions for future research, which we discuss in this section.

Static system properties We believe that ScalaLoci's sound and coherent model for developing distributed systems that enables reasoning about placement and data flows can serve as the foundation for reasoning about overarching properties of the entire distributed system. In preliminary work in that direction, we investigated a type system for tracking latency and making the cost of remote calls explicit, raising the developers' awareness of communication overhead [Weisenburger et al. 2018b]. Further opportunities revolve around the tracking of privacy-related properties using techniques such as information flow control in security type systems [Sabelfeld and Myers 2006].

An additional communication mechanism, which complements ScalaLoci's reactive data flows and remote procedures, are message channels that adhere to multiparty session types [Honda et al. 2008], which specify message-based communication protocols between multiple parties. In ScalaLoci, every party could be represented by a different peer. For programs that type-check using a session type, compliance with the corresponding protocol is statically guaranteed. Traditionally, session types have been investigated in the context of process calculi as opposed to the lambda calculus, which is the foundation for programming languages, including ScalaLoci. In the context of Links [Cooper et al. 2006], binary session types were formalized based on a linear lambda calculus [Fowler et al. 2019].

Fault tolerance Building on our current approach to fault tolerance, another interesting line of research is the formal foundation for a fault tolerance model for general-purpose distributed applications – akin to supervision hierarchies in actor systems – based on event streams – instead of on actors – and the design of language abstractions for fault tolerance that fit the declarative nature of reactive data flow specification.

For reactive systems, existing techniques for achieving tolerance regarding partial failures can be incorporated, such as automatic crash recovery with state restoration and eventually consistent data types [Mogk et al. 2019].

Dynamic placement Another extension to the current language design – which captures placement statically in the type system – is the support for dynamically placing computations. Such an extension to the programming model may keep static placement as the default for static guarantees but programmers could decide that certain functionalities should be able to automatically migrate between hosts to scale or to react to changes of system load or environmental conditions.

Further, an opportunity for enabling dynamic placement to different hosts involves making the mapping of ScalaLoci's *logical* peers to *physical* hosts programmable. Developer-defined placement strategies then make the decision of where peer instances are started and stopped. Another challenge in this research area is the seamless migration of running peer instances between hosts.

Dynamic software updates Further future research may investigate dynamic software updates for ScalaLoci applications, in which case peers can be updated independently of each other during run time. Currently, it is possible to update peers provided there are no changes to the signature of placed values accessed by remote instances. Such compatibility between versions is neither checked nor enforced by ScalaLoci, nor does it support explicit versioning of peers and the update is not seamless in the sense that remote peer instances need to explicitly disconnect from the old version and reconnect to the updated version.

Kramer and Magee [1990] propose an algorithm for putting the components that are part of a software update into a *quiescent* state, which ensures safe dynamic updates, where the system is in a consistent state before and after an update. An improvement over *quiescence* is *tranquillity* [Vandewoude et al. 2007], which uses additional data about the system state to cause less system disruption during software updates.

Data management integration Some multitier languages [Chlipala 2015; Cooper et al. 2006] provide language-level support for database access. ScalaLoci does not specifically support databases. Existing integrated database query languages for Scala, e.g., Slick [Lightbend 2014] or Quill [Ioffe 2015], however, are compatible and can be used in ScalaLoci applications. Integrating database queries into the cross-host data flow is still an open problem. Live data views, which continuously update the result of a query [Mitschke et al. 2014], could pave the way for integration.

Our backend is extensible with new reactive systems, e.g., reactive systems that provide strong consistency guarantees on cross-host value propagation, such as glitch-freedom, can be added [Drechsler et al. 2018].

Trading between consistency and availability is important in distributed systems and weakly consistent data stores have been widely adopted to reduce access

latency at the cost of correctness [Vogels 2009]. So far, ScalaLoci does not provide dedicated language features to this end. Yet, the design of such abstractions may profit from the holistic view ensured by the multitier paradigm.

Bibliography

Agha, Gul. 1986. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA. ISBN: 0-262-01092-5.

Aldrich, Jonathan and Chambers, Craig and Notkin, David. 2002. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, FL, USA. ACM, New York, NY, USA, pages 187–197. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581365.

Alexandrov, Alexander and Bergmann, Rico and Ewen, Stephan and Freytag, Johann-Christoph and Hueske, Fabian and Heise, Arvid and Kao, Odej and Leich, Marcus and Leser, Ulf and Markl, Volker and Naumann, Felix and Peters, Mathias and Rheinländer, Astrid and Sax, Matthias J. and Schelter, Sebastian and Höger, Mareike and Tzoumas, Kostas and Warneke, Daniel. 2014. The Stratosphere platform for big data analytics. *The VLDB Journal* 23 (6). Springer-Verlag, Berlin, Heidelberg, pages 939–964. ISSN: 1066-8888. DOI: 10.1007/s00778-014-0357-y.

AlTurki, Musab and Meseguer, José. 2010. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems*. RTRTS '10. Longyearbyen, Norway, pages 26–45. DOI: 10.4204/EPTCS.36.2.

Andión, José M. and Arenaz, Manuel and Bodin, François and Rodríguez, Gabriel and Touriño, Juan. 2016. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming* 44 (3). Kluwer Academic Publishers, Boston, MA, USA, pages 620–643. ISSN: 0885-7458. DOI: 10.1007/s10766-015-0362-9.

Apache Software Foundation. 2011a. Apache Flink: Powered by Flink. http://flink.apache.org/poweredby.html (visited on April 1, 2020).

Apache Software Foundation. 2011b. Storm. http://storm.apache.org/ (visited on April 1, 2020).

Armstrong, Joe. 2010. Erlang. *Communications of the ACM* 53 (9). ACM, New York, NY, USA, pages 68–75. ISSN: 0001-0782. DOI: 10.1145/1810891.1810910.

Atkinson, Malcolm P. and Buneman, O. Peter. 1987. Types and persistence in database programming languages. *ACM Computing Surveys* 19 (2). ACM, New York, NY, USA, pages 105–170. ISSN: 0360-0300. DOI: 10.1145/62070.45066.

Baeten, J. C. M. 2005. A brief history of process algebra. *Theoretical Computer Science* 335 (2–113). Elsevier Science Publishers Ltd., Essex, UK, pages 131–146. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2004.07.036.

Baker Jr., Henry C. and Hewitt, Carl. 1977. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. ACM, New York, NY, USA, pages 55–59. ISBN: 978-1-4503-7874-1. DOI: 10.1145/800228.806932.

Balat, Vincent. 2006. Ocsigen: Typing web interaction with Objective Caml. In *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, OR, USA. ACM, New York, NY, USA, pages 84–94. ISBN: 1-59593-483-9. DOI: 10.1145/1159876.1159889.

Baltopoulos, Ioannis G. and Gordon, Andrew D. 2009. Secure compilation of a multi-tier web language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. Savannah, GA, USA. ACM, New York, NY, USA, pages 27–38. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481866.

Banken, Herman and Meijer, Erik and Gousios, Georgios. 2018. Debugging data flows in reactive programs. In *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden. ACM, New York, NY, USA, pages 752–763. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180156.

Baumgärtner, Lars and Höchst, Jonas and Lampe, Patrick and Mogk, Ragnar and Sterz, Artur and Weisenburger, Pascal and Mezini, Mira and Freisleben, Bernd. 2019. Smart street lights and mobile citizen apps for resilient communication in a digital city. In *Proceedings of the 2019 IEEE Global Humanitarian Technology Conference*. GHTC '19. Seattle, WA, USA. IEEE Press, Piscataway, NJ, USA. ISBN: 978-1-7281-1781-2. DOI: 10.1109/GHTC46095.2019.9033134.

Berry, Gérard and Gonthier, Georges. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19 (2). Elsevier North-Holland, Inc., Amsterdam, Netherlands, pages 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V.

Berry, Gérard and Nicolas, Cyprien and Serrano, Manuel. 2011. HipHop: A synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*. PLASTIC '11. Portland, OR, USA. ACM, New York, NY, USA, pages 49–56. ISBN: 978-1-4503-1171-7. DOI: 10.1145/2093328.2093337.

Bjornson, Joel and Tayanovskyy, Anton and Granicz, Adam. 2010. Composing reactive GUIs in F# using WebSharper. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*. IFL '10. Alphen aan den Rijn, Netherlands. Springer-Verlag, Berlin, Heidelberg, pages 203–216. ISBN: 978-3-642-24275-5. DOI: 10.1007/978-3-642-24276-2_13.

Black, Andrew P. and Hutchinson, Norman C. and Jul, Eric and Levy, Henry M. 2007. The development of the Emerald programming language. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, Article 11. HOPL III. San Diego, California. ACM, New York, NY, USA, 51 pages. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238855.

Blöcher, Marcel and Eichholz, Matthias and Weisenburger, Pascal and Eugster, Patrick and Mezini, Mira and Salvaneschi, Guido. 2019. GRASS: Generic reactive application-specific scheduling. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '19. Athens, Greece. ACM, New York, NY, USA, pages 21–30. ISBN: 978-1-4503-6986-2. DOI: 10.1145/3358503.3361274.

Boer, Frank de and Serbanescu, Vlad and Hähnle, Reiner and Henrio, Ludovic and Rochas, Justine and Din, Crystal Chang and Johnsen, Einar Broch and Sirjani, Marjan and Khamespanah, Ehsan and Fernandez-Reyes, Kiko and Yang, Albert Mingkun. 2017. A survey of active object languages. *ACM Computing Surveys* 50 (5), Article 76. ACM, New York, NY, USA, 39 pages. ISSN: 0360-0300. DOI: 10.1145/3122848.

Boudol, Gérard and Luo, Zhengqin and Rezk, Tamara and Serrano, Manuel. 2012. Reasoning about web applications: An operational semantics for Hop. *ACM Transactions on Programming Languages and Systems* 34 (2), Article 10. ACM, New York, NY, USA, 40 pages. ISSN: 0164-0925. DOI: 10.1145/2220365.2220369.

Bračevac, Oliver and Erdweg, Sebastian and Salvaneschi, Guido and Mezini, Mira. 2016. CPL: A core language for cloud computing. In *Proceedings of the 15th International Conference on Modularity*. MODULARITY '16. Málaga, Spain. ACM, New York, NY, USA, pages 94–105. ISBN: 978-1-4503-3995-7. DOI: 10.1145/2889443.2889452.

Bracha, Gilad and Ahé, Peter von der and Bykov, Vassili and Kashai, Yaron and Maddox, William and Miranda, Eliot. 2010. Modules as objects in Newspeak. In *Proceedings of the 24th European Conference on Object-Oriented Programming*. ECOOP '10. Maribor, Slovenia. Springer-Verlag, Berlin, Heidelberg, pages 405–428. ISBN: 978-3-642-14106-5. DOI: 10.1007/978-3-642-14107-2_20.

Bracha, Gilad and Cook, William. 1990. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA/ECOOP '90. Ottawa, ON, Canada. ACM, New York, NY, USA, pages 303–311. ISBN: 0-89791-411-2. DOI: 10.1145/97945.97982.

Bracha, Gilad and Odersky, Martin and Stoutamire, David and Wadler, Philip. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented*

Programming, Systems, Languages, and Applications. OOPSLA '98. Vancouver, BC, Canada. ACM, New York, NY, USA, pages 183–200. ISBN: 1-58113-005-8. DOI: 10.1145/286936.286957.

Breitbart, Jens. 2009. CuPP – A framework for easy CUDA integration. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IPDPS '09. Rome, Italy. IEEE Computer Society, Washington, DC, USA. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5160937.

Brown, Travis. 2015a. circe: Semi-automatic derivation. http://circe.github.io/circe/codecs/semiauto-derivation.html#jsoncodec (visited on April 1, 2020).

Brown, Travis. 2015b. circe: Yet another JSON library for Scala. http://circe.github.io/circe/ (visited on April 1, 2020).

Burckhardt, Sebastian and Fähndrich, Manuel and Leijen, Daan and Wood, Benjamin P. 2012. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP '12. Beijing, China. Springer-Verlag, Berlin, Heidelberg, pages 283–307. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_14.

Burmako, Eugene. 2013a. Macro paradise: Now a compiler plugin for 2.10.x, features quasiquotes and macro annotations. http://groups.google.com/forum/#!msg/scala-language/C7Pm6ab1sPs/4-o-7JtQfCgJ (visited on April 1, 2020).

Burmako, Eugene. 2013b. Scala macros: Let our powers combine! On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, Article 3. SCALA '13. Montpellier, France. ACM, New York, NY, USA, 10 pages. ISBN: 978-1-4503-2064-1. DOI: 10.1145/2489837.2489840.

Buschmann, Frank and Meunier, Regine and Rohnert, Hans and Sommerlad, Peter and Stal, Michael. 1996. Pattern-Oriented Software Architecture: A System of Patterns. Volume 1. John Wiley & Sons. ISBN: 978-0-471-95869-7.

Bykov, Sergey and Geller, Alan and Kliot, Gabriel and Larus, James R. and Pandya, Ravi and Thelin, Jorgen. 2011. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Article 16. SOCC '11. Cascais, Portugal. ACM, New York, NY, USA, 14 pages. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038932.

Calcagno, Cristiano and Taha, Walid and Huang, Liwen and Leroy, Xavier. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. GPCE '03. Erfurt, Germany. Springer-Verlag, Berlin, Heidelberg, pages 57–76. ISBN: 978-3-540-20102-1. DOI: 10.1007/978-3-540-39815-8_4.

Calus, Ben and Reynders, Bob and Devriese, Dominique and Noorman, Job and Piessens, Frank. 2017. FRP IoT modules as a Scala DSL. In *Proceedings of the*

4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems. REBLS '17. Vancouver, BC, Canada. ACM, New York, NY, USA, pages 15–20. ISBN: 978-1-4503-5515-5. DOI: 10.1145/3141858.3141861.

Carbone, Marco and Montesi, Fabrizio. 2013. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy. ACM, New York, NY, USA, pages 263–274. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429101.

Carbone, Paris and Katsifodimos, Asterios and Ewen, Stephan and Markl, Volker and Haridi, Seif and Tzoumas, Kostas. 2015. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin* 38 (4), pages 28–38. http://sites.computer.org/debull/A15dec/p28.pdf (visited on April 1, 2020).

Cardelli, Luca and Gordon, Andrew D. 1998. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*. FoSSaCS '98. Lisbon, Portugal. Springer-Verlag, Berlin, Heidelberg, pages 140–155. ISBN: 978-3-540-64300-5. DOI: 10.1007/BFb0053547.

Cardellini, Valeria and Grassi, Vincenzo and Lo Presti, Francesco and Nardelli, Matteo. 2016. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*. DEBS '16. Irvine, CA, USA. ACM, New York, NY, USA, pages 69–80. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933312.

Carreton, Andoni Lombide and Mostinckx, Stijn and Van Cutsem, Tom and De Meuter, Wolfgang. 2010. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*. TOOLS '10. Málaga, Spain. Springer-Verlag, Berlin, Heidelberg, pages 41–60. ISBN: 978-3-642-13952-9. DOI: 10.1007/978-3-642-13953-6_3.

Carzaniga, Antonio and Rosenblum, David S. and Wolf, Alexander L. 2001. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 19 (3). ACM, New York, NY, USA, pages 332–383. ISSN: 0734-2071. DOI: 10.1145/380749.380767.

Cave, Andrew and Ferreira, Francisco and Panangaden, Prakash and Pientka, Brigitte. 2014. Fair reactive programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, CA, USA. ACM, New York, NY, USA, pages 361–372. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535881.

Chambers, Craig and Raniwala, Ashish and Perry, Frances and Adams, Stephen and Henry, Robert R. and Bradshaw, Robert and Weizenbaum, Nathan. 2010.

FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, ON, Canada. Association for Computing Machinery, New York, NY, USA, pages 363–375. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806638.

Chandra, Satish and Saraswat, Vijay and Sarkar, Vivek and Bodik, Rastislav. 2008. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '08. Salt Lake City, UT, USA. ACM, New York, NY, USA, pages 11–22. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345211.

Charles, Philippe and Grothoff, Christian and Saraswat, Vijay and Donawa, Christopher and Kielstra, Allan and Ebcioglu, Kemal and Praun, Christoph von and Sarkar, Vivek. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA. ACM, New York, NY, USA, pages 519–538. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852.

Cheney, James and Hinze, Ralf. 2003. First-Class Phantom Types. Technical report. Cornell University. http://hdl.handle.net/1813/5614 (visited on April 1, 2020).

Cherniack, Mitch and Balakrishnan, Hari and Balazinska, Magdalena and Carney, Donald and Çetintemel, Ugur and Xing, Ying and Zdonik, Stan. 2003. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*. CIDR '03. Asilomar, CA, USA. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p23.pdf (visited on April 1, 2020).

Chintapalli, Sanket and Dagit, Derek and Evans, Bobby and Farivar, Reza and Graves, Thomas and Holderbaugh, Mark and Liu, Zhuo and Nusbaum, Kyle and Patil, Kishorkumar and Peng, Boyang Jerry and Poulosky, Paul. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. IPDPSW '16. Chicago, IL, USA, pages 1789–1792. DOI: 10.1109/IPDPSW.2016.138.

Chlipala, Adam. 2015. Ur/Web: A simple model for programming the web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India. ACM, New York, NY, USA, pages 153–165. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2677004.

Chong, Stephen and Liu, Jed and Myers, Andrew C. and Qi, Xin and Vikram, K. and Zheng, Lantian and Zheng, Xin. 2007a. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review* 41 (6). ACM, New York, NY, USA, pages 31–44. ISSN: 0163-5980. DOI: 10.1145/1323293.1294265.

Chong, Stephen and Vikram, K. and Myers, Andrew C. 2007b. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*, Article 1. SS '07. Boston, MA, USA. USENIX Association, Berkeley, CA, USA, 16 pages. http://usenix.org/events/sec07/tech/full_papers/chong/chong.pdf (visited on April 1, 2020).

Cooper, Ezra and Lindley, Sam and Wadler, Philip and Yallop, Jeremy. 2006. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO '06. Amsterdam, Netherlands. Springer-Verlag, Berlin, Heidelberg, pages 266–296. ISBN: 978-3-540-74791-8. DOI: 10.1007/978-3-540-74792-5_12.

Cooper, Ezra and Lindley, Sam and Wadler, Philip and Yallop, Jeremy. 2008. The essence of form abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. APLAS '08. Bangalore, India. Springer-Verlag, Berlin, Heidelberg, pages 205–220. ISBN: 978-3-540-89329-5. DOI: 10.1007/978-3-540-89330-1_15.

Cooper, Ezra E. K. and Wadler, Philip. 2009. The RPC calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. PPDP '09. Coimbra, Portugal. ACM, New York, NY, USA, pages 231–242. ISBN: 978-1-60558-568-0. DOI: 10.1145/1599410.1599439.

Cooper, Gregory H. and Krishnamurthi, Shriram. 2006. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*. ESOP '06. Vienna, Austria. Springer-Verlag, Berlin, Heidelberg, pages 294–308. ISBN: 978-3-540-33095-0. DOI: 10.1007/11693024_20.

Coq Development Team. 2016. The Coq proof assistant, version 8.11.0. http://coq.inria.fr/ (visited on April 1, 2020).

Crnkovic, Ivica and Sentilles, Severine and Aneta, Vulgarakis and Chaudron, Michel R. V. 2011. A classification framework for software component models. *IEEE Trans. Softw. Eng.* 37 (5). IEEE Press, Piscataway, NJ, USA, pages 593–615. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.83.

Cruz-Filipe, Luís and Montesi, Fabrizio. 2016. Choreographies in practice. In *Proceedings of the 36th IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. FORTE '16. Heraklion, Greece. Springer-Verlag, Berlin, Heidelberg, pages 114–123. ISBN: 978-3-319-39569-2. DOI: 10.1007/978-3-319-39570-8_8.

Cugola, Gianpaolo and Margara, Alessandro. 2013. Deployment strategies for distributed complex event processing. *Computing* 95 (2). Springer-Verlag, Berlin, Heidelberg, pages 129–156. ISSN: 0010-485X. DOI: 10.1007/s00607-012-0217-9.

Czaplicki, Evan and Chong, Stephen. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA. ACM, New York, NY, USA, pages 411–422. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462161.

De Koster, Joeri and Van Cutsem, Tom and De Meuter, Wolfgang. 2016. 43 years of actors: A taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE '16. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 31–40. ISBN: 978-1-4503-4639-9. DOI: 10.1145/3001886.3001890.

De Wael, Mattias and Marr, Stefan and De Fraine, Bruno and Van Cutsem, Tom and De Meuter, Wolfgang. 2015. Partitioned global address space languages. *ACM Computing Surveys* 47 (4), Article 62. ACM, New York, NY, USA, 27 pages. ISSN: 0360-0300. DOI: 10.1145/2716320.

Dean, Jeffrey and Ghemawat, Sanjay. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51 (1). ACM, New York, NY, USA, pages 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

Dedecker, Jessie and Van Cutsem, Tom and Mostinckx, Stijn and D'Hondt, Theo and De Meuter, Wolfgang. 2006. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP '06. Nantes, France. Springer-Verlag, Berlin, Heidelberg, pages 230–254. ISBN: 978-3-540-35726-1. DOI: 10.1007/11785477_16.

Delaval, Gwenaël and Girault, Alain and Pouzet, Marc. 2008. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '08. Tucson, AZ, USA. ACM, New York, NY, USA, pages 101–110. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375672.

Deursen, Arie van and Klint, Paul. 1998. Little languages: Little maintenance? *Journal of Software Maintenance: Research and Practice* 10 (2), pages 75–92. DOI: 10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

Deursen, Arie van and Klint, Paul and Visser, Joost. 2000. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 35 (6). ACM, New York, NY, USA, pages 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035.

Doeraene, Sébastien. 2013. Scala.js: Type-Directed Interoperability with Dynamically Typed Languages. Technical report. EPFL. http://infoscience.epfl.ch/record/190834 (visited on April 1, 2020).

Drechsler, Joscha and Mogk, Ragnar and Salvaneschi, Guido and Mezini, Mira. 2018. Thread-safe reactive programming. *Proceedings of the ACM on Programming*

Languages 2 (OOPSLA), Article 107. Boston, MA, USA. ACM, New York, NY, USA, 30 pages. DOI: 10.1145/3276477.

Drechsler, Joscha and Salvaneschi, Guido and Mogk, Ragnar and Mezini, Mira. 2014. Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, OR, USA. ACM, New York, NY, USA, pages 361–376. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660240.

Edwards, Jonathan. 2009. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, FL, USA. ACM, New York, NY, USA, pages 925–932. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640058.

Ekblad, Anton and Claessen, Koen. 2014. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden. ACM, New York, NY, USA, pages 79–89. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633367.

Elliott, Conal and Hudak, Paul. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 263–273. ISBN: 0-89791-918-1. DOI: 10.1145/258948.258973.

Elliott, Conal M. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland. ACM, New York, NY, USA, pages 25–36. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596643.

Ericsson. 1987a. Erlang: Academic and historical questions. http://erlang.org/faq/academic.html (visited on April 1, 2020).

Ericsson. 1987b. Erlang: Supervisor behaviour. http://erlang.org/documentation/doc-9.3/doc/design_principles/sup_princ.html (visited on April 1, 2020).

Ernst, Erik and Ostermann, Klaus and Cook, William R. 2006. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, SC, USA. ACM, New York, NY, USA, pages 270–282. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111062.

Eugster, Patrick Th. and Felber, Pascal A. and Guerraoui, Rachid and Kermarrec, Anne-Marie. 2003. The many faces of publish/subscribe. *ACM Computing Surveys* 35 (2). ACM, New York, NY, USA, pages 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.

Fehrenbach, Stefan and Cheney, James. 2019. Language-integrated provenance by trace analysis. In *Proceedings of the 17th ACM SIGPLAN International Symposium on*

Database Programming Languages. DBPL '19. Phoenix, AZ, USA. ACM, New York, NY, USA, pages 74–84. ISBN: 978-1-45036-718-9. DOI: 10.1145/3315507.3330198.

Felleisen, Matthias and Findler, Robert Bruce and Flatt, Matthew and Krishnamurthi, Shriram and Barzilay, Eli and McCarthy, Jay and Tobin-Hochstadt, Sam. 2015. The Racket manifesto. In *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL '15)*. Volume 32. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pages 113–128. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113.

Finkbeiner, Bernd and Klein, Felix and Piskac, Ruzica and Santolucito, Mark. 2017. Vehicle platooning simulations with functional reactive programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*. SCAV '17. Pittsburgh, PA, USA. ACM, New York, NY, USA, pages 43–47. ISBN: 978-1-4503-4976-5. DOI: 10.1145/3055378.3055385.

Fischer, Jeffrey and Majumdar, Rupak and Millstein, Todd. 2007. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '07. Nice, France. ACM, New York, NY, USA, pages 134–143. ISBN: 978-1-59593-620-2. DOI: 10.1145/1244381.1244403.

Foster, Nate and Harrison, Rob and Freedman, Michael J. and Monsanto, Christopher and Rexford, Jennifer and Story, Alec and Walker, David. 2011. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan. ACM, New York, NY, USA, pages 279–291. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034812.

Fournet, Cédric and Gonthier, Georges. 1996. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA. ACM, New York, NY, USA, pages 372–385. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237805.

Fowler, Martin. 2010. Domain Specific Languages. 1st. Addison-Wesley Professional. ISBN: 0-321-71294-3.

Fowler, Simon and Lindley, Sam and Morris, J. Garrett and Decova, Sára. 2019. Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages* 3 (POPL), Article 28. Cascais, Portugal. ACM, New York, NY, USA, 29 pages. ISSN: 2475-1421. DOI: 10.1145/3290341.

Garlan, David and Shaw, Mary. 1994. An Introduction to Software Architecture. Technical report. http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf (visited on April 1, 2020).

Gasiunas, Vaidas and Mezini, Mira and Ostermann, Klaus. 2007. Dependent classes. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada. ACM, New York, NY, USA, pages 133–152. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297038.

Gelernter, David. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7 (1). ACM, New York, NY, USA, pages 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433.

Giallorenzo, Saverio and Montesi, Fabrizio and Peressotti, Marco. 2020. Choreographies as objects. arXiv: 2005.09520.

Gorlatch, Sergei. 2004. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems* 26 (1). ACM, New York, NY, USA, pages 47–56. ISSN: 0164-0925. DOI: 10.1145/963778.963780.

Groenewegen, Danny M. and Hemel, Zef and Kats, Lennart C. L. and Visser, Eelco. 2008. WebDSL: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA Companion '08. Nashville, TN, USA. ACM, New York, NY, USA, pages 779–780. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449858.

Group, Object Management. 1993. The Common Object Request Broker: Architecture and Specification. Wiley-QED. ISBN: 978-0-471-58792-7.

Grust, Torsten and Mayr, Manuel and Rittinger, Jan and Schreiber, Tom. 2009. Ferry – Database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, RI, USA. ACM, New York, NY, USA, pages 1063–1066. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559982.

Guha, Arjun and Jeannin, Jean-Baptiste and Nigam, Rachit and Tangen, Jane and Shambaugh, Rian. 2017. Fission: Secure dynamic code-splitting for JavaScript. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL '17)*. Volume 71, Article 5. Leibniz International Proceedings in Informatics (LIPIcs). Asilomar, CA, USA. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13 pages. ISBN: 978-3-95977-032-3. DOI: 10.4230/LIPIcs.SNAPL.2017.5.

Haller, Philipp and Odersky, Martin. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410 (2–3). Elsevier Science Publishers Ltd., Essex, UK, pages 202–220. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.09.019.

Haller, Philipp and Zaugg, Jason. 2012. Scala Async. http://github.com/scala/async (visited on April 1, 2020).

Haridi, Seif and Van Roy, Peter and Smolka, Gert. 1997. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation*. PASCO '97. Maui, HI, USA. ACM, New York, NY, USA, pages 176–187. ISBN: 0-89791-951-3. DOI: 10.1145/266670.266726.

Hart, Timothy P. 1963. MACRO Definitions for LISP. Technical report. AI Memo 57. Massachusetts Institute of Technology, Artificial Intelligence Project, RLE and MIT Computation Center. http://hdl.handle.net/1721.1/6111 (visited on April 1, 2020).

Haxe Foundation. 2005. Haxe cross-platform toolkit. http://haxe.org (visited on April 1, 2020).

Headley, Kyle. 2018. A DSL embedded in Rust. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. IFL '18. Lowell, MA, USA. ACM, New York, NY, USA, pages 119–126. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310241.

Hemel, Zef and Visser, Eelco. 2011. Declaratively programming the mobile web with Mobl. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, OR, USA. ACM, New York, NY, USA, pages 695–712. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048121.

Hewitt, Carl and Bishop, Peter and Steiger, Richard. 1973. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI '73. Stanford, CA, USA. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pages 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf (visited on April 1, 2020).

Hillerström, Daniel and Lindley, Sam and Atkey, Robert and Sivaramakrishnan, KC. 2017. Continuation passing style for effect handlers. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD '17)*. Volume 84, Article 18. Leibniz International Proceedings in Informatics (LIPIcs). Oxford, UK. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19 pages. ISBN: 978-3-95977-047-7. DOI: 10.4230/LIPIcs.FSCD.2017.18.

Hirschberg, D. S. and Sinclair, J. B. 1980. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM* 23 (11). ACM, New York, NY, USA, pages 627–628. ISSN: 0001-0782. DOI: 10.1145/359024.359029.

Honda, Kohei and Yoshida, Nobuko and Carbone, Marco. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, CA,

USA. ACM, New York, NY, USA, pages 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472.

Hudak, Paul and Courtney, Antony and Nilsson, Henrik and Peterson, John. 2003. Arrows, robots, and functional reactive programming. In. *Advanced Functional Programming: 4th International School (AFP 2002)*. Volume 2638. Lecture Notes in Computer Science. Oxford, UK. Springer-Verlag, Berlin, Heidelberg. ISBN: 978-3-540-40132-2. DOI: 10.1007/978-3-540-44833-4_6.

Huebsch, Ryan and Hellerstein, Joseph M. and Lanham, Nick and Loo, Boon Thau and Shenker, Scott and Stoica, Ion. 2003. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB '03. Berlin, Germany. VLDB Endowment, pages 321–332. ISBN: 0-12-722442-4. DOI: 10.1016/B978-012722442-8/50036-7.

Hunt, Galen C. and Scott, Michael L. 1999. The Coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, LA, USA. USENIX Association, Berkeley, CA, USA, pages 187–200. ISBN: 1-880446-39-1. http://usenix.org/events/osdi99/full_papers/hunt/hunt.pdf (visited on April 1, 2020).

Ioffe, Alexander. 2015. Quill. http://getquill.io/ (visited on April 1, 2020).

Isard, Michael and Yu, Yuan. 2009. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, RI, USA. ACM, New York, NY, USA, pages 987–994. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559962.

Iu, Ming-Yee and Cecchet, Emmanuel and Zwaenepoel, Willy. 2010. JReq: Database queries in imperative languages. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*. CC/ETAPS '10. Paphos, Cyprus. Springer-Verlag, Berlin, Heidelberg, pages 84–103. ISBN: 978-3-642-11969-9. DOI: 10.1007/978-3-642-11970-5_6.

JetBrains. 2009. Kotlin programming language. http://kotlinlang.org (visited on April 1, 2020).

Kendall, Samuel C. and Waldo, Jim and Wollrath, Ann and Wyant, Geoff. 1994. A Note on Distributed Computing. Technical report. Sun Microsystems, Inc.

Kereki, Federico. 2010. Essential GWT: Building for the Web with Google Web Toolkit 2. 1st. Addison-Wesley Professional. ISBN: 978-0-321-70514-3.

Khronos OpenCL Working Group. 2009. The OpenCL specification. In *Proceedings of the 2009 IEEE Hot Chips 21 Symposium*. HCS '09. Stanford, CA, USA. DOI: 10.1109/HOTCHIPS.2009.7478342.

Klepper, Robert and Bock, Douglas. 1995. Third and fourth generation language productivity differences. *Communications of the ACM* 38 (9). ACM, New York, NY, USA, pages 69–79. ISSN: 0001-0782. DOI: 10.1145/223248.223268.

Kloudas, Konstantinos and Mamede, Margarida and Preguiça, Nuno and Rodrigues, Rodrigo. 2015. Pixida: Optimizing data parallel jobs in wide-area data analytics. *Proceedings of the VLDB Endowment* 9 (2). VLDB Endowment, pages 72–83. ISSN: 2150-8097. DOI: 10.14778/2850578.2850582.

Kramer, Jeff and Magee, Jeff. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering* 16 (11). IEEE Press, Piscataway, NJ, USA, pages 1293–1306. ISSN: 0098-5589. DOI: 10.1109/32.60317.

Krishnaswami, Neelakantan R. and Benton, Nick and Hoffmann, Jan. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA. ACM, New York, NY, USA, pages 45–58. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103665.

Lakshmanan, Geetika T. and Li, Ying and Strom, Rob. 2008. Placement strategies for Internet-scale data stream systems. *IEEE Internet Computing* 12 (6). IEEE Educational Activities Department, Piscataway, NJ, USA, pages 50–60. ISSN: 1089-7801. DOI: 10.1109/MIC.2008.129.

Lanese, Ivan and Guidi, Claudio and Montesi, Fabrizio and Zavattaro, Gianluigi. 2008. Bridging the gap between interaction- and process-oriented choreographies. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*. SEFM '08. Cape Town, South Africa. IEEE Computer Society, Washington, DC, USA, pages 323–332. ISBN: 978-0-7695-3437-4. DOI: 10.1109/SEFM.2008.11.

Lawlor, Orion Sky. 2011. Embedding OpenCL in C++ for expressive GPU programming. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. WOLFHPC '11. Tucson, AZ, USA. http://hpc.pnl.gov/conf/wolfhpc/2011/papers/L2011.pdf (visited on April 1, 2020).

Leijen, Daan. 2014. Koka: Programming with row polymorphic effect types. In *Proceedings of the 5th Workshop on Mathematically Structured Functional Programming*. MSFP '14. Grenoble, France, pages 100–126. DOI: 10.4204/EPTCS.153.8.

Leroy, Xavier. 2000. A modular module system. *Journal of Functional Programming* 10 (3). Cambridge University Press, Cambridge, UK, pages 269–303. ISSN: 0956-7968. DOI: 10.1017/S0956796800003683.

Li, Haoyi. 2014. µPickle. http://www.lihaoyi.com/upickle/ (visited on April 1, 2020).

Li, Haoyi. 2015. Pure-Scala CRDTs. http://github.com/lihaoyi/crdt (visited on April 1, 2020).

Li, Haoyi. 2012. ScalaTags. http://www.lihaoyi.com/scalatags/ (visited on April 1, 2020).

Li, Haoyi. 2013. TodoMVC application written in Scala.js. http://github.com/lihaoyi/workbench-example-app (visited on April 1, 2020).

Lightbend. 2009a. Akka. http://akka.io/ (visited on April 1, 2020).

Lightbend. 2016. Akka HTTP. http://doc.akka.io/docs/akka-http/ (visited on April 1, 2020).

Lightbend. 2009b. Akka: Message delivery reliability. http://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html (visited on April 1, 2020).

Lightbend. 2009c. Akka: Watching remote actors. http://doc.akka.io/docs/akka/current/remoting-artery.html#watching-remote-actors (visited on April 1, 2020).

Lightbend. 2009d. Akka: What lifecycle monitoring means. http://doc.akka.io/docs/akka/current/general/supervision.html#what-lifecycle-monitoring-means (visited on April 1, 2020).

Lightbend. 2009e. Akka: What supervision means. http://doc.akka.io/docs/akka/current/general/supervision.html#what-supervision-means (visited on April 1, 2020).

Lightbend. 2007. Play Framework. http://playframework.com/ (visited on April 1, 2020).

Lightbend. 2014. Slick. http://scala-slick.org/ (visited on April 1, 2020).

Lindley, Sam and Morris, J. Garrett. 2017. Lightweight functional session types. In *Behavioural Types: From Theory to Tools*. River Publishers. Chapter 12. ISBN: 978-87-93519-82-4. DOI: 10.13052/rp-9788793519817.

Liskov, Barbara. 1988. Distributed programming in Argus. *Communications of the ACM* 31 (3). ACM, New York, NY, USA, pages 300–312. ISSN: 0001-0782. DOI: 10.1145/42392.42399.

Liskov, Barbara. 1987. Keynote address – Data abstraction and hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA Addendum '87. Orlando, FL, USA. ACM, New York, NY, USA, pages 17–34. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62141.

Lluch Lafuente, Alberto and Nielson, Flemming and Nielson, Hanne Riis. 2015. Discretionary information flow control for interaction-oriented specifications. In. *Logic, Rewriting, and Concurrency*. Volume 9200. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, pages 427–450. ISBN: 978-3-319-23164-8. DOI: 10.1007/978-3-319-23165-5_20.

Luo, Zhengqin and Rezk, Tamara and Serrano, Manuel. 2011. Automated code injection prevention for web applications. In *Proceedings of the 2011 International Conference on Theory of Security and Applications*. TOSCA '11. Saarbrücken, Germany. Springer-Verlag, Berlin, Heidelberg, pages 186–204. ISBN: 978-3-642-27374-2. DOI: 10.1007/978-3-642-27375-9_11.

Luthra, Manisha and Koldehofe, Boris and Weisenburger, Pascal and Salvaneschi, Guido and Arif, Raheel. 2018. TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*. DEBS '18. Hamilton, New Zealand. ACM, New York, NY, USA, pages 136–147. ISBN: 978-1-4503-5782-1. DOI: 10.1145/3210284.3210292.

Maier, Ingo and Odersky, Martin. 2013. Higher-order reactive programming with incremental lists. In *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP '13. Montpellier, France. Springer-Verlag, Berlin, Heidelberg, pages 707–731. ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_29.

Maier, Ingo and Rompf, Tiark and Odersky, Martin. 2010. Deprecating the Observer Pattern. Technical report. EPFL. http://infoscience.epfl.ch/record/148043 (visited on April 1, 2020).

Manolescu, Dragos and Beckman, Brian and Livshits, Benjamin. 2008. Volta: Developing distributed applications by recompiling. *IEEE Software* 25 (5), pages 53–59. ISSN: 0740-7459. DOI: 10.1109/MS.2008.131.

Margara, Alessandro and Salvaneschi, Guido. 2018. On the semantics of distributed reactive programming: The cost of consistency. *IEEE Transactions on Software Engineering* 44 (7). IEEE Press, Piscataway, NJ, USA, pages 689–711. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2833109.

Margara, Alessandro and Salvaneschi, Guido. 2014. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India. ACM, New York, NY, USA, pages 142–153. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611290.

Massie, Matthew L. and Chun, Brent N. and Culler, David E. 2004. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing* 30 (7), pages 817–840. ISSN: 0167-8191. DOI: 10.1016/j.parco.2004.04.001.

Medvidovic, Nenad and Taylor, Richard N. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26 (1). IEEE Press, Piscataway, NJ, USA, pages 70–93. ISSN: 0098-5589. DOI: 10.1109/32.825767.

Meier, René and Cahill, Vinny. 2002. Taxonomy of distributed event-based programming systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. ICDCSW '02. Vienna, Austria. IEEE Computer Society, Washington, DC, USA, pages 585–586. ISBN: 0-7695-1588-6. DOI: 10.1109/ICDCSW.2002.1030833.

Meijer, Erik. 2010. Reactive Extensions (Rx): Curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, MD, USA. ACM, New York, NY, USA. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900173.

Mernik, Marjan and Heering, Jan and Sloane, Anthony M. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys* 37 (4). ACM, New York, NY, USA, pages 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

Meyerovich, Leo A. and Guha, Arjun and Baskin, Jacob and Cooper, Gregory H. and Greenberg, Michael and Bromfield, Aleks and Krishnamurthi, Shriram. 2009. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, FL, USA. ACM, New York, NY, USA, pages 1–20. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640091.

Miller, Heather and Haller, Philipp and Müller, Normen and Boullier, Jocelyn. 2016. Function passing: A model for typed, distributed functional programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 82–97. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986014.

Miller, Heather and Haller, Philipp and Odersky, Martin. 2014. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *Proceedings of the 28th European Conference on Object-Oriented Programming*. ECOOP '14. Uppsala, Sweden. Springer-Verlag, Berlin, Heidelberg, pages 308–333. ISBN: 978-3-662-44201-2. DOI: 10.1007/978-3-662-44202-9_13.

Miller, Mark S. and Tribble, E. Dean and Shapiro, Jonathan. 2005. Concurrency among strangers: Programming in E as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*. TGC '05. Edinburgh, UK. Springer-Verlag, Berlin, Heidelberg, pages 195–229. ISBN: 978-3-540-30007-6. DOI: 10.1007/11580850_12.

Milner, Robin and Morris, Lockwood and Newey, Malcolm. 1975. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of the Conference on Proving and Improving Programs*. Arc-et-Senans, France, pages 371–394.

Mitschke, Ralf and Erdweg, Sebastian and Köhler, Mirko and Mezini, Mira and Salvaneschi, Guido. 2014. i3QL: Language-integrated live data views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, OR, USA. ACM, New York, NY, USA, pages 417–432. ISBN: 978-1-45032-585-1. DOI: 10.1145/2660193.2660242.

Mogk, Ragnar and Baumgärtner, Lars and Salvaneschi, Guido and Freisleben, Bernd and Mezini, Mira. 2018a. Fault-tolerant distributed reactive programming. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP '18)*. Volume 109, Article 1. Leibniz International Proceedings in Informatics (LIPIcs). Amsterdam, Netherlands. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26 pages. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.1.

Mogk, Ragnar and Drechsler, Joscha and Salvaneschi, Guido and Mezini, Mira. 2019. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages* 3 (OOPSLA), Article 144. Athens, Greece. ACM, New York, NY, USA, 29 pages. DOI: 10.1145/3360570.

Mogk, Ragnar and Weisenburger, Pascal and Haas, Julian and Richter, David and Salvaneschi, Guido and Mezini, Mira. 2018b. From debugging towards live tuning of reactive applications. In *2018 LIVE Programming Workshop*. LIVE '18. Boston, MA, USA.

Montesi, Fabrizio. 2014. Kickstarting choreographic programming. In *Proceedings of the 13th International Workshop on Web Services and Formal Methods*. WS-FM '14. Eindhoven, Netherlands. Springer-Verlag, Berlin, Heidelberg, pages 3–10. ISBN: 978-3-319-33611-4. DOI: 10.1007/978-3-319-33612-1_1.

Montesi, Fabrizio and Guidi, Claudio and Zavattaro, Gianluigi. 2014. Service-oriented programming with Jolie. In. *Web Services Foundations*. Springer-Verlag, Berlin, Heidelberg, pages 81–107. ISBN: 978-1-4614-7517-0. DOI: 10.1007/978-1-4614-7518-7_4.

Moors, Adriaan and Rompf, Tiark and Haller, Philipp and Odersky, Martin. 2012. Scala-Virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. PEPM '12. Philadelphia, PA, USA. ACM, New York, NY, USA, pages 117–120. ISBN: 978-1-4503-1118-2. DOI: 10.1145/2103746.2103769.

Murphy VII, Tom and Crary, Karl and Harper, Robert. 2007. Type-safe distributed programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*. TGC '07. Sophia-Antipolis, France. Springer-Verlag, Berlin, Heidelberg, pages 108–123. ISBN: 978-3-540-78662-7. DOI: 10.1007/978-3-540-78663-4_9.

Myter, Florian and Coppieters, Tim and Scholliers, Christophe and De Meuter, Wolfgang. 2016. I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '16. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 1–8. ISBN: 978-1-4503-4644-3. DOI: 10.1145/3001929.3001930.

Nelson, Tim and Ferguson, Andrew D. and Scheer, Michael J. G. and Krishnamurthi, Shriram. 2014. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI '14. Seattle, WA, USA. USENIX Association, Berkeley, CA, USA, pages 519–531. ISBN: 978-1-931971-09-6. http://usenix.org/system/files/conference/nsdi14/nsdi14-paper-nelson.pdf (visited on April 1, 2020).

Neubauer, Matthias and Thiemann, Peter. 2005. From sequential programs to multi-tier applications by program transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, CA, USA. ACM, New York, NY, USA, pages 221–232. ISBN: 978-1-58113-830-6. DOI: 10.1145/1040305.1040324.

Newton, Ryan and Morrisett, Greg and Welsh, Matt. 2007. The Regiment macroprogramming system. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. IPSN '07. Cambridge, Massachusetts, USA. ACM, New York, NY, USA, pages 489–498. ISBN: 978-1-59593-638-7. DOI: 10.1145/1236360.1236422.

Nilsson, Henrik and Courtney, Antony and Peterson, John. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, PA, USA. ACM, New York, NY, USA, pages 51–64. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581695.

Nitzberg, Bill and Lo, Virginia. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24 (8). IEEE Computer Society Press, Washington, DC, USA, pages 52–60. ISSN: 0018-9162. DOI: 10.1109/2.84877.

Nystrom, Nathaniel and Chong, Stephen and Myers, Andrew C. 2004. Scalable extensibility via nested inheritance. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, Canada. ACM, New York, NY, USA, pages 99–115. ISBN: 1-58113-831-8. DOI: 10.1145/1028976.1028986.

Nystrom, Nathaniel and Qi, Xin and Myers, Andrew C. 2006. J&: Nested intersection for scalable software composition. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*.

OOPSLA '06. Portland, OR, USA. ACM, New York, NY, USA, pages 21–36. ISBN: 1-59593-348-4. DOI: 10.1145/1167473.1167476.

Odersky, Martin and Blanvillain, Olivier and Liu, Fengyun and Biboudis, Aggelos and Miller, Heather and Stucki, Sandro. 2017. Simplicitly: Foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages* 2 (POPL), Article 42. Paris, France. ACM, New York, NY, USA, 29 pages. DOI: 10.1145/3158130.

Odersky, Martin and Martres, Guillaume and Petrashko, Dmitry. 2016. Implementing higher-kinded types in dotty. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA '16. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 51–60. ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.2998400.

Odersky, Martin and Stucki, Nicolas. 2018. Macros: The plan for Scala 3. http://www.scala-lang.org/blog/2018/04/30/in-a-nutshell.html#future-macros (visited on April 1, 2020).

Odersky, Martin and Zenger, Matthias. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA. ACM, New York, NY, USA, pages 41–57. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094815.

Oliveira, Bruno C. d. S. and Moors, Adriaan and Odersky, Martin. 2010. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, NV, USA. ACM, New York, NY, USA, pages 341–360. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869489.

Olston, Christopher and Reed, Benjamin and Srivastava, Utkarsh and Kumar, Ravi and Tomkins, Andrew. 2008. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, BC, Canada. ACM, New York, NY, USA, pages 1099–1110. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376726.

Ongaro, Diego and Ousterhout, John. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC '14. Philadelphia, PA. USENIX Association, Berkeley, CA, USA, pages 305–319. ISBN: 978-1-931971-10-2. http://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf (visited on April 1, 2020).

Park, David. 1981. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. Karlsruhe, Germany. Springer-Verlag, Berlin, Heidelberg, pages 167–183. ISBN: 978-3-540-10576-3. DOI: 10.1007/BFb0017309.

Perez, Ivan and Nilsson, Henrik. 2017. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages* 1 (ICFP), Article 2. Oxford, UK. ACM, New York, NY, USA, 27 pages. DOI: 10.1145/3110246.

Perry, Dewayne E. and Wolf, Alexander L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17 (4). ACM, New York, NY, USA, pages 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884.

Persson, Per and Angelsmark, Ola. 2015. Calvin – Merging Cloud and IoT. *Procedia Computer Science*. ANT/SEIT '15 52 (The 6th International Conference on Ambient Systems, Networks and Technologies, the 5th International Conference on Sustainable Energy Information Technology). London, UK. Elsevier, pages 210–217. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.059.

Philips, Laure and De Koster, Joeri and De Meuter, Wolfgang and De Roover, Coen. 2018. Search-based tier assignment for optimising offline availability in multi-tier web applications. *The Art, Science, and Engineering of Programming* 2 (2), Article 3. AOSA, Inc, 29 pages. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/3.

Pietzuch, Peter and Ledlie, Jonathan and Shneidman, Jeffrey and Roussopoulos, Mema and Welsh, Matt and Seltzer, Margo. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*. ICDE '06. Atlanta, GA, USA. IEEE Computer Society, Washington, DC, USA, pages 49–60. ISBN: 0-7695-2570-9. DOI: 10.1109/ICDE.2006.105.

Pietzuch, Peter R. and Bacon, Jean. 2002. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*. ICDCSW '02. Vienna, Austria. IEEE Computer Society, Washington, DC, USA, pages 611–618. ISBN: 0-7695-1588-6. DOI: 10.1109/ICDCSW.2002.1030837.

Preda, Mila Dalla and Gabbrielli, Maurizio and Giallorenzo, Saverio and Lanese, Ivan and Mauro, Jacopo. 2017. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science* 13 (2). DOI: 10.23638/LMCS-13(2:1)2017.

Puripunpinyo, Hussachai. 2014. Play framework with Scala.js showcase. http://github.com/hussachai/play-scalajs-showcase/ (visited on April 1, 2020).

Radanne, Gabriel and Vouillon, Jérôme. 2018. Tierless web programming in the large. In *Companion Proceedings of the The Web Conference 2018*. WWW Companion '18. Lyon, France. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, pages 681–689. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3185953.

Radanne, Gabriel and Vouillon, Jérôme and Balat, Vincent. 2016. Eliom: A core ML language for tierless web programming. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*. APLAS '16. Hanoi, Vietnam. Springer-Verlag, Berlin, Heidelberg, pages 377–397. ISBN: 978-3-319-47957-6. DOI: 10.1007/978-3-319-47958-3_20.

Rajchenbach-Teller, David and Sinot, Franois-Régis. 2010. Opa: Language support for a sane, safe and secure web. In *Proceedings of the OWASP AppSec Research*. Stockholm, Sweden. http://owasp.org/www-pdf-archive/OWASP_AppSec_Research_2010_OPA_by_Rajchenbach-Teller.pdf (visited on April 1, 2020).

Reactive Streams Initiative. 2014. Reactive Streams. http://www.reactive-streams.org/ (visited on April 1, 2020).

Rellermeyer, Jan S. and Alonso, Gustavo and Roscoe, Timothy. 2007. R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. Middleware '07. Newport Beach, CA, USA. Springer-Verlag, Berlin, Heidelberg, pages 1–20. ISBN: 978-3-540-76777-0. DOI: 10.1007/978-3-540-76778-7_1.

Reynders, Bob and Piessens, Frank and Devriese, Dominique. 2020. Gavial: Programming the web with multi-tier FRP. *The Art, Science, and Engineering of Programming* 4 (3), Article 6. AOSA, Inc, 32 pages. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/6.

Richard-Foy, Julien and Barais, Olivier and Jézéquel, Jean-Marc. 2013. Efficient high-level abstractions for web programming. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. GPCE '13. Indianapolis, IN, USA. ACM, New York, NY, USA, pages 53–60. ISBN: 978-1-4503-2373-4. DOI: 10.1145/2517208.2517227.

Rompf, Tiark and Odersky, Martin. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*. GPCE '10. Eindhoven, Netherlands. ACM, New York, NY, USA, pages 127–136. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868314.

Rossberg, Andreas and Dreyer, Derek. 2013. Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems* 35 (1), Article 2. ACM, New York, NY, USA, 84 pages. ISSN: 0164-0925. DOI: 10.1145/2450136.2450137.

Sabelfeld, Andrei and Myers, Andrew C. 2006. Language-based information-flow security. *IEEE J.Sel. A. Commun.* 21 (1). IEEE Press, Piscataway, NJ, USA, pages 5–19. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.806121.

Sabin, Miles. 2011. shapeless: Generic programming for Scala. http://github.com/milessabin/shapeless (visited on April 1, 2020).

Salvaneschi, Guido and Amann, Sven and Proksch, Sebastian and Mezini, Mira. 2014a. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '14. Hong Kong, China. ACM, New York, NY, USA, pages 564–575. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635895.

Salvaneschi, Guido and Hintz, Gerold and Mezini, Mira. 2014b. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland. ACM, New York, NY, USA, pages 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083.

Salvaneschi, Guido and Mezini, Mira. 2016. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas. ACM, New York, NY, USA, pages 796–807. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884815.

Salvaneschi, Guido and Mezini, Mira. 2014. Towards reactive programming for object-oriented applications. In. *Transactions on Aspect-Oriented Software Development XI*. Volume 8400. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, pages 227–261. ISBN: 978-3-642-55098-0. DOI: 10.1007/978-3-642-55099-7_7.

Santoro, Nicola. 2006. Design and analysis of distributed algorithms. In. Volume 56. Wiley InterScience. Chapter 3.8. ISBN: 978-0-471-71997-7. DOI: 10.1002/0470072644.

Schmidt, Joachim W. and Matthes, Florian. 1994. The DBPL project: Advances in modular database programming. *Information Systems* 19 (2). Elsevier Science Publishers Ltd., Essex, UK, pages 121–140. ISSN: 0306-4379. DOI: 10.1016/0306-4379(94)90007-8.

Seefried, Sean and Chakravarty, Manuel and Keller, Gabriele. 2004. Optimising embedded DSLs using Template Haskell. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering*. GPCE '04. Vancouver, BC, Canada. Springer-Verlag, Berlin, Heidelberg, pages 186–205. ISBN: 978-3-540-23580-4. DOI: 10.1007/978-3-540-30175-2_10.

Serrano, Manuel and Gallesio, Erick and Loitsch, Florian. 2006. Hop, a language for programming the web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA Companion '06. Portland, OR, USA. ACM, New York, NY, USA.

Serrano, Manuel and Prunet, Vincent. 2016. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP '16. Nara, Japan. ACM, New York, NY, USA, pages 180–192. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951916.

Serrano, Manuel and Queinnec, Christian. 2010. A multi-tier semantics for Hop. *Higher-Order and Symbolic Computation* 23 (4). Kluwer Academic Publishers, Hingham, MA, USA, pages 409–431. ISSN: 1388-3690. DOI: 10.1007/s10990-010-9061-9.

Sewell, Peter and Leifer, James J. and Wansbrough, Keith and Nardelli, Francesco Zappa and Allen-Williams, Mair and Habouzit, Pierre and Vafeiadis, Viktor. 2005. Acute: High-level programming language design for distributed computation. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia. ACM, New York, NY, USA, pages 15–26. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086370.

Shapiro, Marc and Preguiça, Nuno and Baquero, Carlos and Zawirski, Marek. 2011a. A comprehensive study of convergent and commutative replicated data types, page 47. http://hal.inria.fr/inria-00555588 (visited on April 1, 2020).

Shapiro, Marc and Preguiça, Nuno and Baquero, Carlos and Zawirski, Marek. 2011b. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS '11. Grenoble, France. Springer-Verlag, Berlin, Heidelberg, pages 386–400. ISBN: 978-3-642-24549-7. DOI: 10.1007/978-3-642-24550-3_29.

Sheard, Tim and Jones, Simon Peyton. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, PA, USA. ACM, New York, NY, USA, pages 1–16. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581691.

Soloway, Elliot and Ehrlich, Kate. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 10 (5). IEEE Press, Piscataway, NJ, USA, pages 595–609. ISSN: 0098-5589. DOI: 10.1109/TSE.1984.5010283.

Spinellis, Diomidis. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 56 (1). Elsevier Science Publishers Ltd., Essex, UK, pages 91–99. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(00)00089-3.

Steen, Maarten and Tanenbaum, Andrew S. 2016. A brief introduction to distributed systems. *Computing* 98 (10). Springer-Verlag, Berlin, Heidelberg, pages 967–1009. ISSN: 0010-485X. DOI: 10.1007/s00607-016-0508-7.

Strack, Isaac. 2012. Getting Started with Meteor.js JavaScript Framework. 1st. Packt Publishing. ISBN: 978-0-321-70514-3.

Szyperski, Clemens. 2002. Component Software: Beyond Object-Oriented Programming. 2nd. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN: 0-201-74572-0.

Taha, Walid and Sheard, Tim. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '97. Amsterdam, Netherlands.

ACM, New York, NY, USA, pages 203–217. ISBN: 0-89791-917-3. DOI: 10.1145/258993.259019.

Thekkath, Chandramohan A. and Levy, Henry M. and Lazowska, Edward D. 1994. Separating data and control transfer in distributed operating systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, CA, USA. ACM, New York, NY, USA, pages 2–11. ISBN: 0-89791-660-3. DOI: 10.1145/195473.195481.

Thorup, Kresten Krab. 1997. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-oriented Programming*. ECOOP '97. Jyväskylä, Finland. Springer-Verlag, Berlin, Heidelberg, pages 444–471. ISBN: 978-3-540-63089-0. DOI: 10.1007/BFb0053390.

Thywissen, John A. and Peters, Arthur Michener and Cook, William R. 2016. Implicitly distributing pervasively concurrent programs: Extended abstract. In *Proceedings of the 1st Workshop on Programming Models and Languages for Distributed Computing*, Article 1. PMLDC '16. Rome, Italy. ACM, New York, NY, USA, 4 pages. ISBN: 978-1-4503-4775-4. DOI: 10.1145/2957319.2957370.

Tilevich, Eli and Smaragdakis, Yannis. 2002. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*. ECOOP '02. London, UK. Springer-Verlag, Berlin, Heidelberg, pages 178–204. ISBN: 978-3-540-43759-8. DOI: 10.1007/3-540-47993-7_8.

TodoMVC Development Team. 2011. TodoMVC. http://todomvc.com/ (visited on April 1, 2020).

Tomlinson, Chris and Kim, Won and Scheevel, Mark and Singh, Vineet and Will, Becky and Agha, Gul. 1988. Rosette: An object-oriented concurrent systems architecture. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. OOPSLA/ECOOP Companion '88. San Diego, CA, USA. ACM, New York, NY, USA, pages 91–93. ISBN: 0-89791-304-3. DOI: 10.1145/67386.67410.

Torgersen, Mads. 2007. Querying in C#: How language integrated query (LINQ) works. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA Companion '07. Montreal, QC, Canada. ACM, New York, NY, USA, pages 852–853. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297922.

Truffaut, Julien. 2014. Monocle: Lens. http://julien-truffaut.github.io/Monocle/optics/lens.html#lens-generation (visited on April 1, 2020).

Van Roy, Peter and Haridi, Seif and Brand, Per and Smolka, Gert and Mehl, Michael and Scheidhauer, Ralf. 1997. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems* 19 (5). ACM, New York, NY, USA, pages 804–851. ISSN: 0164-0925. DOI: 10.1145/265943.265972.

Vandewoude, Yves and Ebraert, Peter and Berbers, Yolande and D'Hondt, Theo. 2007. Tranquility: A low disruptive alternative to Quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering* 33 (12). IEEE Press, Piscataway, NJ, USA, pages 856–868. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70733.

Varela, Carlos and Agha, Gul. 2001. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices* 36 (12). ACM, New York, NY, USA, pages 20–34. ISSN: 0362-1340. DOI: 10.1145/583960.583964.

Venners, Bill. 2009. ScalaTest. http://www.scalatest.org/ (visited on April 1, 2020).

Viñas, Moisés and Bozkus, Zeki and Fraguela, Basilio B. 2013. Exploiting heterogeneous parallelism with the heterogeneous programming library. *Journal of Parallel and Distributed Computing* 73 (12). Academic Press, Inc., Boston, MA, USA, pages 1627–1638. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2013.07.013.

Vogels, Werner. 2009. Eventually consistent. *Communications of the ACM* 52 (1). ACM, New York, NY, USA, pages 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432.

W3C WS-CDL Working Group. 2005. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/ (visited on April 1, 2020).

Weisenburger, Pascal. 2019. Demo: Developing distributed systems with ScalaLoci. In *Demo Track of the 3th International Conference on the Art, Science, and Engineering of Programming*. Programming Demos '19. Genoa, Italy.

Weisenburger, Pascal. 2016. Multitier reactive abstractions. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion '16. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 18–20. ISBN: 978-1-4503-4437-1. DOI: 10.1145/2984043.2984051.

Weisenburger, Pascal. 2015. retypecheck. http://github.com/stg-tud/retypecheck (visited on April 1, 2020).

Weisenburger, Pascal and Köhler, Mirko and Salvaneschi, Guido. 2018a. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2 (OOPSLA), Article 129. Boston, MA, USA. ACM, New York, NY, USA, 30 pages. ISSN: 2475-1421. DOI: 10.1145/3276499.

Weisenburger, Pascal and Luthra, Manisha and Koldehofe, Boris and Salvaneschi, Guido. 2017. Quality-aware runtime adaptation in complex event processing. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '17. Buenos Aires, Argentina. IEEE Press,

Piscataway, NJ, USA, pages 140–151. ISBN: 978-1-5386-1550-8. DOI: 10.1109/SEAMS.2017.10.

Weisenburger, Pascal and Reinhard, Tobias and Salvaneschi, Guido. 2018b. Static latency tracking with placement types. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA/ECOOP Companion '18. Amsterdam, Netherlands. ACM, New York, NY, USA, pages 34–36. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236486.

Weisenburger, Pascal and Salvaneschi, Guido. 2020. Implementing a language for distributed systems: Choices and experiences with type level and macro programming in Scala. *The Art, Science, and Engineering of Programming* 4 (3), Article 17. AOSA, Inc, 29 pages. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/17.

Weisenburger, Pascal and Salvaneschi, Guido. 2019a. Multitier modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)*. Volume 134, Article 3. Leibniz International Proceedings in Informatics (LIPIcs). London, UK. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29 pages. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.3.

Weisenburger, Pascal and Salvaneschi, Guido. 2018. Multitier reactive programming with ScalaLoci. In *5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '18. Boston, MA, USA.

Weisenburger, Pascal and Salvaneschi, Guido. 2016. Towards a comprehensive multitier reactive language. In *3rd International Workshop on Reactive and Event-Based Languages and Systems*. REBLS '16. Amsterdam, Netherlands.

Weisenburger, Pascal and Salvaneschi, Guido. 2019b. Tutorial: Developing distributed systems with multitier programming. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*. DEBS '19. Darmstadt, Germany. ACM, New York, NY, USA, pages 203–204. ISBN: 978-1-4503-6794-3. DOI: 10.1145/3328905.3332465.

Weisenburger, Pascal and Wirth, Johannes and Salvaneschi, Guido. 2020. A survey of multitier programming. *ACM Computing Surveys* 53 (4), Article 81. ACM, New York, NY, USA, 35 pages. ISSN: 0360-0300. DOI: 10.1145/3397495.

Weiser, Mark. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, CA, USA. IEEE Press, Piscataway, NJ, USA, pages 439–449. ISBN: 0-89791-146-6.

Westin, Marcus. 2010. Fun: A programming language for the realtime web. http://marcuswest.in/essays/fun-intro/ (visited on April 1, 2020).

Wienke, Sandra and Springer, Paul and Terboven, Christian and Mey, Dieter an. 2012. OpenACC – First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*. Euro-Par '12. Rhodes

Island, Greece. Springer-Verlag, Berlin, Heidelberg, pages 859–870. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_85.

Wile, David. 2004. Lessons learned from real DSL experiments. *Science of Computer Programming* 51 (3). Elsevier North-Holland, Inc., Amsterdam, Netherlands, pages 265–290. ISSN: 0167-6423. DOI: 10.1016/j.scico.2003.12.006.

Wollrath, Ann and Riggs, Roger and Waldo, Jim. 1996. A distributed object model for the Java™ system. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies*. COOTS '96. Toronto, ON, Canada. USENIX Association, Berkeley, CA, USA. http://www.usenix.org/legacy/publications/library/proceedings/coots96/full_papers/wollrath/wollrath.ps (visited on April 1, 2020).

Wong, Limsoon. 2000. Kleisli, a functional query system. *Journal of Functional Programming* 10 (1). Cambridge University Press, Cambridge, UK, pages 19–56. ISSN: 0956-7968. DOI: 10.1017/S0956796899003585.

Wright, Andrew K. and Felleisen, Matthias. 1994. A syntactic approach to type soundness. *Information and Computation* 115 (1). Academic Press, Inc., Duluth, MN, USA, pages 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093.

Yang, Fan and Gupta, Nitin and Gerner, Nicholas and Qi, Xin and Demers, Alan and Gehrke, Johannes and Shanmugasundaram, Jayavel. 2007. A unified platform for data driven web applications with automatic client-server partitioning. In *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, AB, Canada. ACM, New York, NY, USA, pages 341–350. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242619.

Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J. and Shenker, Scott and Stoica, Ion. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI '12. San Jose, CA, USA. USENIX Association, Berkeley, CA, USA. http://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf (visited on April 1, 2020).

Zdancewic, Steve and Zheng, Lantian and Nystrom, Nathaniel and Myers, Andrew C. 2002. Secure program partitioning. *ACM Transactions on Computer Systems* 20 (3). ACM, New York, NY, USA, pages 283–328. ISSN: 0734-2071. DOI: 10.1145/566340.566343.

Zhang, Irene and Szekeres, Adriana and Van Aken, Dana and Ackerman, Isaac and Gribble, Steven D. and Krishnamurthy, Arvind and Levy, Henry M. 2014. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI '14. Broomfield, CO. USENIX Association, Berkeley, CA, USA,

pages 97–112. ISBN: 978-1-931971-16-4. http://www.usenix.org/system/files/conference/osdi14/osdi14-paper-zhang.pdf (visited on April 1, 2020).

Zhang, Yizhou and Loring, Matthew C. and Salvaneschi, Guido and Liskov, Barbara and Myers, Andrew C. 2015. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA. ACM, New York, NY, USA, pages 436–445. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738008.

Zhang, Yizhou and Myers, Andrew C. 2017. Familia: Unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages* 1 (OOPSLA), Article 70. Vancouver, BC, Canada. ACM, New York, NY, USA, 31 pages. DOI: 10.1145/3133894.

Zhong, Sean and Kasravi, Kam and Wang, Huafeng and Zhang, Manu and Jiang, Weihua. 2014. GearPump – Real-Time Streaming Engine Using Akka. Technical report. Intel Big Data Technology Team. http://downloads.typesafe.com/website/presentations/GearPump_Final.pdf (visited on April 1, 2020).

Zhou, Yongluan and Ooi, Beng Chin and Tan, Kian-Lee and Wu, Ji. 2006. Efficient dynamic operator placement in a locally distributed continuous query system. In *Proceedings of the 2006 Confederated International Conference "On the Move to Meaningful Internet Systems": CoopIS, DOA, GADA, and ODBASE*. OTM '06. Montpellier, France. Springer-Verlag, Berlin, Heidelberg, pages 54–71. ISBN: 978-3-540-48287-1. DOI: 10.1007/11914853_5.

List of Figures

List of Tables

List of Listings

A Proofs

A.1 Type Soundness

For proving type soundness, we use the following lemmas and definitions introduced in Section 5.4 on page 100.

First, we show that a transmitted remote value as modeled by ζ (Listing 5.2 on page 94) can be typed on the local peer:

Lemma 3 (Transmission)**.** *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi; \Delta; \Gamma; P_1 \vdash v : U$ and $t = \zeta(P_1, \vartheta, v, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi; \Delta'; \Gamma'; P_0 \vdash t : U$ for any Δ' and Γ'.*

Proof. By induction on v. □

Second, we show that aggregation modeled by φ (Listing 5.2 on page 94) yields the type given by Φ:

Lemma 4 (Aggregation)**.** *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi; \Delta; \Gamma; P_1 \vdash v : U$ and $t = \varphi(P_0, P_1, \vartheta, v, U)$ and $T = \Phi(P_0, P_1, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi; \Delta'; \Gamma'; P_0 \vdash t : T$ for any Δ' and Γ'.*

Proof. By case analysis on the tie multiplicity $P_0 \leftrightarrow P_1$ and, in the case $P_0 \overset{*}{\rightarrowtail} P_1$, by induction on ϑ and the transmission lemma. □

Next, we provide a definition of typability for the reactive system ρ. We denote with $\mathrm{refs}(\rho)$ the set of all reactive references allocated by ρ:

Definition 3. A reactive system ρ is well-typed with respect to typing contexts Δ, Γ and a reactive typing Ψ, written $\Psi; \Delta; \Gamma \vdash \rho$, iff $\mathrm{refs}(\rho) = \mathrm{dom}(\Psi)$ and $\Psi; \Delta; \Gamma; P \vdash \rho(r) : T$ with $\Psi(r) = \mathsf{Var}\, T \,\mathsf{on}\, P$ or $\Psi(r) = \mathsf{Signal}\, T \,\mathsf{on}\, P$ for every $r \in \mathrm{refs}(\rho)$.

We further use following lemmas, which are standard (thus, their proofs are omitted):

Lemma 5 (Reactive System Store Weakening in t-terms)**.** *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\Psi' \supseteq \Psi$, then $\Psi'; \Delta; \Gamma; P \vdash t : T$.*

Lemma 6 (Reactive System Store Weakening in s-terms)**.** *If $\Psi; \Delta \vdash s$ and $\Psi' \supseteq \Psi$, then $\Psi'; \Delta \vdash s$.*

Lemma 7 (Substitution in t-terms). *If* $\Psi; \Delta; \Gamma, x{:}\, T_0; P \vdash t_1 : T_1$ *and* $\Psi; \varnothing; \varnothing; P \vdash t_0 : T_0$ *then* $\Psi; \Delta; \Gamma; P \vdash [x \mapsto t_0]t_1 : T_1$.

Lemma 8 (Substitution in s-terms). *If* $\Psi; \Delta, x{:}\, T \,\mathsf{on}\, P \vdash s$ *and* $\Psi; \varnothing; \varnothing; P \vdash t : T$ *then* $\Psi; \Delta \vdash [x \mapsto t]s$.

Lemma 9 (Inversion of the Typing Relation).

1. *If* $\Psi; \Delta; \Gamma; P \vdash x : T$, *then* $x{:}\, T \in \Gamma$ *or* $x{:}\, T \,\mathsf{on}\, P \in \Delta$.
2. *If* $\Psi; \Delta; \Gamma; P \vdash \lambda x{:}\, T_1.\, t : T$, *then* $T = T_1 \to T_2$ *for some* T_2 *with* $\Psi; \Delta; \Gamma, x{:}\, T_1; P \vdash t : T_2$.
3. *If* $\Psi; \Delta; \Gamma; P \vdash t_1\, t_2 : T$, *then there is some type* T_1 *such that* $\Psi; \Delta; \Gamma \vdash t_1 : T_1 \to T$ *and* $\Psi; \Delta; \Gamma \vdash t_2 : T_1$.
4. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{cons}\, t_0\, t_1 : T$, *then* $T = \mathsf{List}\, T_1$ *for some* T_1 *with* $\Psi; \Delta; \Gamma; P \vdash t_0 : T_1$ *and* $\Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{List}\, T_1$.
5. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{nil}\,\mathsf{of}\, T_1 : T$, *then* $T = \mathsf{List}\, T_1$.
6. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{some}\, t : T$, *then* $T = \mathsf{Option}\, T_1$ *for some* T_1 *with* $\Psi; \Delta; \Gamma; P \vdash t : T_1$.
7. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{none}\,\mathsf{of}\, T_1 : T$, *then* $T = \mathsf{Option}\, T_1$.
8. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{unit} : T$, *then* $T = \mathsf{Unit}$.
9. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{asLocal}\, t{:}\, T_1 \,\mathsf{on}\, P_1 : T$, *then* $T = \Phi(P, P_1, T_1)$ *and* $\Psi; \Delta; \varnothing; P_1 \vdash t : T_1$ *and* $P \leftrightarrow P_1$.
10. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{asLocal}\, t_1{:}\, T_1 \,\mathsf{on}\, P_1 \,\mathsf{from}\, t_2 : T$, *then* $T = T_1$ *and* $\Psi; \Delta; \varnothing; P_1 \vdash t_1 : T_1$ *and* $\Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{Remote}\, P_1$ *and* $P \leftrightarrow P_1$.
11. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{asLocal}\,\mathsf{run}\, x{:}\, T_2 = t_2\,\mathsf{in}\, t_1 : T_1 \,\mathsf{on}\, P_1 : T$, *then* $T = \Phi(P, P_1, T_1)$ *and* $\Psi; \Delta; \Gamma; P \vdash t_2 : T_2$ *and* $\Psi; \Delta; x{:}\, T_2; P_1 \vdash t_1 : T_1$ *and* $P \leftrightarrow P_1$.
12. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{asLocal}\,\mathsf{run}\, x{:}\, T_2 = t_2\,\mathsf{in}\, t_1 : T_1 \,\mathsf{on}\, P_1 \,\mathsf{from}\, t_3 : T$, *then* $T = T_1$ *and* $\Psi; \Delta; \Gamma; P \vdash t_2 : T_2$ *and* $\Psi; \Delta; x{:}\, T_2; P_1 \vdash t_1 : T_1$ *and* $\Psi; \Delta; \Gamma; P \vdash t_3 : \mathsf{Remote}\, P_1$ *and* $P \leftrightarrow P_1$.
13. *If* $\Psi; \Delta; \Gamma; P \vdash \vartheta : T$, *then* $\mathsf{Remote}\, P_1$ *with* $\vartheta \subseteq \mathcal{I}^{[P_1]}$.
14. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{signal}\, t : T$, *then* $T = \mathsf{Signal}\, T_1$ *for some* T_1 *with* $\Psi; \Delta; \Gamma; P \vdash t : T_1$.
15. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{var}\, t : T$, *then* $T = \mathsf{Var}\, T_1$ *for some* T_1 *with* $\Psi; \Delta; \Gamma; P \vdash t : T_1$.

16. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{now}\, t : T$, *then* $T = \mathsf{Signal}\, T_0$ *or* $T = \mathsf{Var}\, T_0$ *for some* T_0 *with* $\Psi; \Delta; \Gamma; P \vdash t : T_1$.

17. *If* $\Psi; \Delta; \Gamma; P \vdash \mathsf{set}\, t_1 := t_2 : T$, *then* $T = \mathsf{Unit}$ *and* $\Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Var}\, T_1$ *and* $\Psi; \Delta; \Gamma; P \vdash t_2 : T_1$ *for some* T_1.

18. *If* $\Psi; \Delta; \Gamma; P \vdash r : T$, *then* $T\, \mathsf{on}\, P = \Psi(r)$ *and* $T = \mathsf{Signal}\, T_0$ *or* $T = \mathsf{Var}\, T_0$ *for some* T_0.

Lemma 10 (Canonical Forms)**.**

1. *If* v *is a value of type* $T_1 \to T_2$, *then* $v = \lambda x{:}\, T_1.\, t$.

2. *If* v *is a value of type* Unit, *then* $v = \mathsf{unit}$.

3. *If* v *is a value of type* $\mathsf{Option}\, T$, *then* $v = \mathsf{none}\, \mathsf{of}\, T$ *or* $v = \mathsf{some}\, v_0$.

4. *If* v *is a value of type* $\mathsf{List}\, T$, *then* $v = \mathsf{nil}\, \mathsf{of}\, T$ *or* $v = \mathsf{cons}\, v_0\, v_1$.

5. *If* v *is a value of type* $\mathsf{Remote}\, T$, *then* $v = \vartheta$.

6. *If* v *is a value of type* $\mathsf{Signal}\, T$ *or* $\mathsf{Var}\, T$, *then* $v = r$.

We prove type soundness based on the usual notion of progress and preservation [Wright and Felleisen 1994], meaning that well-typed programs do not get stuck during evaluation. We first formulate progress and preservation for terms t:

Theorem 8 (Progress on t-terms)**.** *Suppose* t *is a closed, well-typed term (that is,* $\Psi; \varnothing; \varnothing; P \vdash t : T$ *for some* T, P *and* Ψ*). Then either* t *is a value or else, for any* ϑ *and any reactive system* ρ *such that* $\Psi; \varnothing; \varnothing \vdash \rho$, *there is some term* t', *some* ϑ' *and some reactive system* ρ' *with* $\vartheta{:}\, P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$.

Proof. Induction on a derivation of $t{:}\, T$.

Case T-VAR: $t = x$

Cannot happen. The term t is closed.

Case T-APP: $t = t_1\, t_2 \quad t_1{:}\, T_2 \to T_1 \quad t_2{:}\, T_2$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation, and likewise t_2. If t_1 can take a step or if t_1 is a value and t_2 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If both t_1 and t_2 are values, then from the canonical forms lemma we obtain that t_1 has the form $\lambda x{:}\, T_2.\, t_3$, and so E-APP applies to t for any ϑ and ρ.

Case T-ABS: $t = \lambda x{:}\, T_1.\, t$

The term t is a value.

Case T-UNIT: $t = \mathsf{unit}$

The term t is a value.

Case T-CONS: $t = \mathsf{cons}\, t_1\, t_2 \quad t_1 : T \quad t_2 : \mathsf{List}\, T$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation, and likewise t_2. If t_1 can take a step or if t_1 is a value and t_2 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If both t_1 and t_2 are values, then t is value.

Case T-NIL: $t = \mathsf{nil}\, \mathsf{of}\, T$

The term t is a value.

Case T-SOME: $t = \mathsf{some}\, t_1 \quad t_1 : T$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation. If t_1 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If t_1 is a value, then t is value.

Case T-NONE: $t = \mathsf{none}\, \mathsf{of}\, T$

The term t is a value.

Case T-PEER: $t = \vartheta$

The term t is a value.

Case T-ASLOCAL: $t = \mathsf{asLocal}\, t_1 : T_1\, \mathsf{on}\, P_1 \quad t_1 : T_1 \quad P_0 \leftrightarrow P_1$
$T_0 = \Phi(P_0, P_1, T_1) \quad P = P_0$

By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation. If t_1 can take a step, then E-REMOTE applies to t for any ϑ and ρ. If t_1 is a value v, then E-ASLOCAL applies to t for any ϑ and ρ and there is a t' with $t' = \varphi(P_0, P_1, \mathcal{I}^{[P_1]}, v, T_1)$ by Definition 1.

Case T-ASLOCALFROM: $t = \mathsf{asLocal}\, t_0 : T\, \mathsf{on}\, P_1\, \mathsf{from}\, t_1 \quad t_0 : T \quad P_0 \leftrightarrow P_1$
$t_1 : \mathsf{Remote}\, P_1 \quad P = P_0$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation, and likewise t_0. If t_1 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If t_1 is a value, then from the canonical forms lemma we obtain that $t_1 = \vartheta$. If t_1 is a value and t_0 can take a step, then E-REMOTEFROM applies to t for any ϑ' and ρ. If both t_1 and t_0 are values, then E-ASLOCALFROM applies to t for any ϑ' and ρ.

Case T-BLOCK: $t = \mathsf{asLocal\ run}\ x = t_0 : T_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1 \quad t_0 : T_0 \quad t_1 : T_1$
$P_0 \leftrightarrow P_1 \quad T_2 = \Phi(P_0, P_1, T_1) \quad P = P_0$

By the induction hypothesis, either t_0 is a value or else it can make a step of evaluation. If t_0 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If t_0 is a value, then E-BLOCK applies to t for any ϑ and ρ.

Case T-BLOCKFROM: $t = \mathsf{asLocal\ run}\ x : T_0 = t_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1\ \mathsf{from}\ t_2 \quad t_0 : T_0$
$t_1 : T_1 \quad P_0 \leftrightarrow P_1 \quad t_2 : \mathsf{Remote}\ P_1 \quad P = P_0$

By the induction hypothesis, either t_2 is a value or else t_2 can make a step of evaluation, and likewise t_0. If t_2 can take a step or if t_2 is a value and t_1 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If both t_2 and t_0 are values, then from the canonical forms lemma we obtain that $t_2 = \vartheta$, and so E-BLOCK applies to t for any ϑ' and ρ.

Case T-REACTIVE: $t = r$

The term t is a value.

Case T-SIGNAL: $t = \mathsf{signal}\ t_1 \quad t_1 : T$

E-SIGNAL applies to t for any ϑ and ρ.

Case T-SOURCEVAR: $t = \mathsf{var}\ t_1 \quad t_1 : T$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation. If t_1 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If t_1 is a value, then E-SOURCEVAR applies to t for any ϑ and ρ.

Case T-NOW: $t = \mathsf{now}\ t_1 \quad t_1 : T_1 \quad T_1 = \mathsf{Signal}\ T_0 \ \vee\ T_1 = \mathsf{Var}\ T_0$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation. If t_1 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If t_1 is a value, then from the canonical forms lemma we obtain that t_1 is a reactive reference r. Using the inversion lemma, we can destruct the typing derivation for $\Psi; \varnothing; \varnothing; P \vdash r : T_1$ yielding $r \in \mathrm{dom}(\Psi)$. From this and $\Psi; \varnothing; \varnothing \vdash \rho$, we conclude that $r \in \mathrm{refs}(\rho)$, and so E-NOW applies to t for ρ and any ϑ.

Case T-SET: $t = \mathsf{set}\ t_1 := t_2 \quad t_1 : \mathsf{Var}\ T \quad t_2 : T$

By the induction hypothesis, either t_1 is a value or else t_1 can make a step of evaluation, and likewise t_2. If t_1 can take a step or if t_1 is a value and t_2 can take a step, then E-CONTEXT applies to t for any ϑ and ρ. If both t_1 and t_2 are values, then from the canonical forms lemma we obtain that t_1 is a reactive reference r. Using the inversion lemma, we can destruct the

typing derivation for $\Psi; \varnothing; \varnothing; P \vdash r : \mathsf{Var}\, T$ yielding $r \in \mathrm{dom}(\Psi)$. From this and $\Psi; \varnothing; \varnothing \vdash \rho$, we conclude that $r \in \mathrm{refs}(\rho)$, and so E-SET applies to t for ρ and any ϑ. □

Theorem 9 (Preservation on t-terms)**.** *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\Psi; \Delta; \Gamma \vdash \rho$ and $\vartheta: P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ with $\vartheta \in \mathcal{I}^{[P]}$, then $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. Induction on a derivation of $t: T$.

Case T-VAR: $t = x$

Cannot happen. There are no evaluation rules for variables.

Case T-APP: $t = t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_2 \to T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : T_2$
$T = T_1$

There are three cases by which $\vartheta: P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value, (ii) E-CONTEXT if t_1 is a value and t_2 is not a value and (iii) E-APP if t_1 and t_2 are values.

(i) E-CONTEXT: $\vartheta: P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = t_1'\, t_2$

From the induction hypothesis, the weakening lemma and T-APP, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta: P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho' \quad t' = v\, t_2'$

Similar to (i).

(iii) E-APP: $t_1 = \lambda x{:}\, T_2.\, t_3 \quad t_2 = v \quad t' = [x \mapsto v]t_3 \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. Using the inversion lemma, we can destruct the typing derivation for $\Psi; \Delta; \Gamma; P \vdash \lambda x{:}\, T_2.\, t_3 : T_2 \to T_1$ yielding $\Psi; \Delta; \Gamma, x{:}\, T_2; P \vdash t_3 : T_1$. From this and the substitution lemma, we conclude $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-ABS: $t = \lambda x{:}\, T_1.\, t$

Cannot happen. The term t is already a value.

Case T-UNIT: $t = \mathsf{unit}$

Cannot happen. The term t is already a value.

Case T-CONS: $t = \mathsf{cons}\, t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{List}\, T_1$
$T = \mathsf{List}\, T_1$

There are two cases by which $\vartheta: P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value and (ii) E-CONTEXT if t_1 is a value and t_2 is not a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{cons}\, t_1'\, t_2$

From the induction hypothesis, the weakening lemma and T-CONS, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta\colon P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho' \quad t' = \mathsf{cons}\, v\, t_2'$

Similar to (i).

Case T-NIL: $t = \mathsf{nil\ of}\ T_1$

Cannot happen. The term t is already a value.

Case T-SOME: $t = \mathsf{some}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Option}\, T_1$

There is only a single case by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:

E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{some}\, t_1'$

From the induction hypothesis and T-SOME, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

Case T-NONE: $t = \mathsf{none\ of}\ T_1$

Cannot happen. The term t is already a value.

Case T-PEER: $t = \vartheta$

Cannot happen. The term t is already a value.

Case T-ASLOCAL: $t = \mathsf{asLocal}\, t_1 \colon T_1\ \mathsf{on}\ P_1 \quad \Psi; \Delta; \varnothing; P_1 \vdash t_1 : T_1$
$P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

There are two cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived: (i) E-REMOTE if t_1 is not a value and (ii) E-ASLOCAL if t_1 is a value.

(i) E-REMOTE: $\mathcal{I}^{[P_1]}\colon P_1 \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{asLocal}\, t_1' \colon T_1\ \mathsf{on}\ P_1$

From the induction hypothesis and T-ASLOCAL, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-ASLOCAL: $t_1 = v \quad t' = \varphi(P_0, P_1, \mathcal{I}^{[P_1]}, v, T) \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. From this and the aggregation lemma, we conclude $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-ASLOCALFROM: $t = \mathsf{asLocal}\, t_0 \colon T\ \mathsf{on}\ P_1\ \mathsf{from}\ t_1$
$\Psi; \Delta; \varnothing; P_1 \vdash t_0 : T_1 \quad P \leftrightarrow P_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Remote}\, P_1 \quad T = T_1$

There are three cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value, (ii) E-REMOTEFROM if t_1 is a peer instances value and t_0 is not a value and (iii) E-ASLOCALFROM if t_1 is peer instances value and t_0 is a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{asLocal}\, t_0\colon T\, \mathsf{on}\, P_1\, \mathsf{from}\, t_1'$

From the induction hypothesis, the weakening lemma and T-ASLOCALFROM, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-REMOTEFROM: $t_1 = \vartheta'' \quad \vartheta''\colon P_1 \rhd t_0; \rho \xrightarrow{\vartheta'} t_0'; \rho'$
$t' = \mathsf{asLocal}\, t_0'\colon T\, \mathsf{on}\, P_1\, \mathsf{from}\, p$

Similar to (i).

(iii) E-ASLOCALFROM: $t_0 = v \quad t_1 = \vartheta' \quad t' = \zeta(P_1, \vartheta', v, T) \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. From this and the transmission lemma, we conclude $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-BLOCK: $t = \mathsf{asLocal\ run}\, x\colon T_0 = t_0\, \mathsf{in}\, t_1 : T_1\, \mathsf{on}\, P_1$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x\colon T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

There are two cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_0 is not a value and (ii) E-BLOCK if t_0 is a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_0; \rho \xrightarrow{\vartheta'} t_0'; \rho'$
$t' = \mathsf{asLocal\ run}\, x\colon T_0 = t_0'\, \mathsf{in}\, t_1 : T_1\, \mathsf{on}\, P_1$

From the induction hypothesis, the weakening lemma and T-BLOCK, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-BLOCK: $t_0 = v \quad t'' = \zeta(P_0, \vartheta, v, T_0)$
$t' = \mathsf{asLocal}\, [x \mapsto t'']t_1 : T_1\, \mathsf{on}\, P_1 \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. From this and the substitution lemma and the transmission lemma and T-ASLOCAL, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-BLOCKFROM: $t = \mathsf{asLocal\ run}\, x\colon T_0 = t_0\, \mathsf{in}\, t_1 : T_1\, \mathsf{on}\, P_1\, \mathsf{from}\, t_2$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x\colon T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1$
$\Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{Remote}\, P_1$

There are three cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_2 is not a value, (ii) E-CONTEXT if t_2 is a peer instances value and t_0 is not a value and (iii) E-BLOCKFROM if t_2 is peer instances value and t_0 is a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho'$
$t' = \mathsf{asLocal}\ \mathsf{run}\ x\colon T_0 = t_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1\ \mathsf{from}\ t_2'$

From the induction hypothesis, the weakening lemma and T-BLOCKFROM, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-CONTEXT: $t_2 = \vartheta' \quad \vartheta\colon P \rhd t_0; \rho \xrightarrow{\vartheta''} t_0'; \rho'$
$t' = \mathsf{asLocal}\ \mathsf{run}\ x\colon T_0 = t_0'\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1\ \mathsf{from}\ p$

Similar to (i).

(iii) E-BLOCKFROM: $t_0 = v \quad t'' = \zeta(P_0, \vartheta, v, T_0) \quad t_2 = \vartheta'$
$t' = \mathsf{asLocal}\ [x \mapsto t'']t_1 : T_1\ \mathsf{on}\ P_1\ \mathsf{from}\ p \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. From this and the substitution lemma and the transmission lemma and T-ASLOCALFROM, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-REACTIVE: $t = r$

Cannot happen. The term t is already a value.

Case T-SIGNAL: $t = \mathsf{signal}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\, T_1$

There is only a single case by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:

E-SIGNAL: $t' = r \quad \rho' = (\rho, r \mapsto t) \quad r \notin \mathrm{dom}(\rho)$

For $\Psi' = (\Psi, r \mapsto \mathsf{Signal}\, T_1)$, we conclude from T-REACTIVE that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$.

Case T-SOURCEVAR: $t = \mathsf{var}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Var}\, T_1$

There are two cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value and (ii) E-SOURCEVAR if t_1 is a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{var}\, t_1'$

From the induction hypothesis and T-SOURCEVAR, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-VAR: $t_1 = v \quad t' = r \quad \rho' = (\rho, r \mapsto t) \quad r \notin \mathrm{dom}(\rho)$

For $\Psi' = (\Psi, r \mapsto \mathsf{Var}\, T_1)$, we conclude from T-REACTIVE that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$.

Case T-NOW: $t = \mathsf{now}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\, T_1 \ \vee\ T = \mathsf{Var}\, T_1$

There are two cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value and (ii) E-NOW if t_1 is a value.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{now}\, t_1'$

From the induction hypothesis and T-NOW, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-NOW: $t_1 = r \quad t' = \rho(r) \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \Gamma \vdash \rho'$. Using the inversion lemma, we can destruct the typing derivation for $\Psi; \Delta; \Gamma; P \vdash r : T_1$ yielding $\Psi(r) = T_1$. From this and $\Psi; \Delta; \Gamma \vdash \rho$, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$.

Case T-SET: $t = \mathsf{set}\, t_1 := t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Var}\, T_1$
$\Psi; \Gamma; \Gamma; P \vdash t_2 : T_1 \quad T = \mathsf{Unit}$

There are three cases by which $\vartheta\colon P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ can be derived:
(i) E-CONTEXT if t_1 is not a value, (ii) E-CONTEXT if t_1 is a value and t_2 is not a value and (iii) E-SET if t_1 and t_2 are values.

(i) E-CONTEXT: $\vartheta\colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho' \quad t' = \mathsf{set}\, t_1' := t_2$

From the induction hypothesis, the weakening lemma and T-SET, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta\colon P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho' \quad t' = \mathsf{set}\, v := t_2'$

Similar to (i).

(iii) E-SET: $t_1 = r \quad t_2 = v \quad t' = \mathsf{unit} \quad \rho' = [r \mapsto v]\rho$

Using the inversion lemma, we can destruct the typing derivation for $\Psi; \Delta; \Gamma; P \vdash r : \mathsf{Var}\, T_1$ yielding $\Psi(r) = \mathsf{Var}\, T_1$. From this and $\Psi; \Delta; \Gamma \vdash \rho$ and T-UNIT, we conclude that $\Psi'; \Delta; \Gamma; P \vdash t' : T$ and $\Psi'; \Delta; \Gamma \vdash \rho'$ for $\Psi = \Psi'$. □

Based on progress and preservation for terms t, we prove type soundness for whole programs s:

Theorem 10 (Progress on s-terms)**.** *Suppose s is a closed, well-typed term (that is, $\Psi; \varnothing \vdash s$ for some Ψ). Then either s is a value or else, for any reactive system ρ such that $\Psi; \varnothing; \varnothing \vdash \rho$, there is some term s', some ϑ and some reactive system ρ' with $s; \rho \xrightarrow{\vartheta} s'; \rho'$.*

Proof. Case analysis on the derivation of s.

Case T-PLACED: $s = \mathsf{placed}\, x{:}T\, \mathsf{on}\, P = t\, \mathsf{in}\, s_0 \quad t : T$

By the progress lemma for t-terms, either t is a value or else it can make a step of evaluation. If t can take a step, then E-PLACED applies to s. If t is a value, then E-PLACEDVAL applies to s.

Case T-END: $s = \mathsf{unit}$

The term s is a value. □

Theorem 11 (Preservation on s-terms). *If $\Psi; \Delta \vdash s$ and $\Psi; \Delta; \varnothing \vdash \rho$ and $s; \rho \xrightarrow{\vartheta} s'; \rho'$, then $\Psi'; \Delta \vdash s'$ and $\Psi'; \Delta; \varnothing \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. Case analysis on the derivation of s.

Case T-PLACED: $s = \mathsf{placed}\ x{:}\,T\ \mathsf{on}\ P = t\ \mathsf{in}\ s_0 \quad \Psi; \Delta; \varnothing; P \vdash t : T$
$\Psi; \Delta, x{:}\,T\ \mathsf{on}\ P \vdash s_0$

There are two cases by which $s; \rho \xrightarrow{\vartheta} s'; \rho'$ can be derived: (i) E-PLACED if t is not a value and (ii) E-PLACEDVAL if t is a value.

(i) E-PLACED: $\mathcal{I}^{[P]}{:}\,P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho' \quad s' = \mathsf{placed}\ x{:}\,T\ \mathsf{on}\ P = t'\ \mathsf{in}\ s_0$

From the preservation lemma for t-terms, the weakening lemma and T-PLACED, we conclude that $\Psi'; \Delta \vdash s'$ and $\Psi'; \Delta; \varnothing \vdash \rho'$ for some $\Psi' \supseteq \Psi$.

(ii) E-PLACEDVAL: $t = v \quad s' = [x \mapsto v]s_0 \quad \rho = \rho'$

We assume $\Psi' = \Psi$, thus $\Psi'; \Delta; \varnothing \vdash \rho'$. From this and the substitution lemma and T-ASLOCAL, we conclude that $\Psi'; \Delta \vdash s'$.

Case T-END: $s = \mathsf{unit}$

Cannot happen. The term s is already a value. □

A.2 Placement Soundness

We prove placement soundness for the core calculus. We show that we can statically reason about the peer on which code is executed, i.e., that the peer context P in which a term t is type-checked matches the peer type P of the peer instances ϑ on which t is evaluated. The type system is sound for a placement if the code placed on a peer P is executed on peer instances of peer type P:

Theorem 12 (Static Placement on t-terms). *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\vartheta{:}\,P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$ with $\vartheta \subseteq \mathcal{I}^{[P]}$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i{:}\,P_i' \rhd t_i; \rho_i \xrightarrow{\vartheta''} t_i'; \rho_i'$ holds $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$ and $P_i = P_i'$.*

Proof. Case analysis on the derivation of $t{:}\,T$.

Case T-VAR: $t = x$

Cannot happen. There are no evaluation rules for variables.

Case T-APP: $t = t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_2 \to T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : T_2$
$T = T_1$

There are two cases by which $\vartheta_i \colon P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived: (i) E-CONTEXT if t_1 is not a value and (ii) E-CONTEXT if t_1 is a value and t_2 is not a value.

(i) E-CONTEXT: $\vartheta \colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta \colon P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho'$

Similar to (i).

Case T-ABS: $t = \lambda x \colon T_1 . \, t$

Cannot happen. The term t is already a value.

Case T-UNIT: $t = \mathsf{unit}$

Cannot happen. The term t is already a value.

Case T-CONS: $t = \mathsf{cons}\, t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{List}\, T_1$
$T = \mathsf{List}\, T_1$

There are two cases by which $\vartheta_i \colon P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived: (i) E-CONTEXT if t_1 is not a value and (ii) E-CONTEXT if t_1 is a value and t_2 is not a value.

(i) E-CONTEXT: $\vartheta \colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta \colon P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho'$

Similar to (i).

Case T-NIL: $t = \mathsf{nil}\ \mathsf{of}\ T_1$

Cannot happen. The term t is already a value.

Case T-SOME: $t = \mathsf{some}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Option}\, T_1$

There is only a single case by which $\vartheta_i \colon P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-CONTEXT: $\vartheta \colon P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-NONE: $t = \mathsf{none}\ \mathsf{of}\ T_1$

Cannot happen. The term t is already a value.

Case T-PEER: $t = \vartheta$

Cannot happen. The term t is already a value.

Case T-ASLOCAL: $t = \mathsf{asLocal}\, t_1 : T_1 \,\mathsf{on}\, P_1 \quad \Psi; \Delta; \varnothing; P_1 \vdash t_1 : T_1$
$P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

There is only a single case by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-REMOTE: $\mathcal{I}^{[P_1]} : P_1 \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \mathcal{I}^{[P_1]}$ and $P_i' = P_i = P_1$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-ASLOCALFROM: $t = \mathsf{asLocal}\, t_0 : T \,\mathsf{on}\, P_1 \,\mathsf{from}\, t_1$
$\Psi; \Delta; \varnothing; P_1 \vdash t_0 : T_1 \quad P \leftrightarrow P_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Remote}\, P_1 \quad T = T_1$

There are two cases by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived: (i) E-CONTEXT if t_1 is not a value and (ii) E-REMOTEFROM if t_1 is a peer instances value and t_0 is not a value.

(i) E-CONTEXT: $\vartheta : P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

(ii) E-REMOTEFROM: $t_1 = \vartheta'' \quad \vartheta'' : P_1 \rhd t_0; \rho \xrightarrow{\vartheta'} t_0'; \rho'$

We obtain $\vartheta_i = \vartheta''$ and $P_i' = P_i = P_1$. Using the inversion lemma, we can destruct the typing derivation for $\Psi; \Delta; \Gamma; P \vdash \vartheta'' : \mathsf{Remote}\, P_1$ yielding $\vartheta'' \subseteq \mathcal{I}^{[(]} P_1)$. From this, we conclude that $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-BLOCK: $t = \mathsf{asLocal}\, \mathsf{run}\, x : T_0 = t_0 \,\mathsf{in}\, t_1 : T_1 \,\mathsf{on}\, P_1$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x : T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

There is only a single case by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-CONTEXT: $\vartheta : P \rhd t_0; \rho \xrightarrow{\vartheta'} t_0'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-BLOCKFROM: $t = \mathsf{asLocal}\, \mathsf{run}\, x : T_0 = t_0 \,\mathsf{in}\, t_1 : T_1 \,\mathsf{on}\, P_1 \,\mathsf{from}\, t_2$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x : T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1$
$\Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{Remote}\, P_1$

There are two cases by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived: (i) E-CONTEXT if t_2 is not a value and (ii) E-CONTEXT if t_2 is a peer instances value and t_0 is not a value.

(i) E-CONTEXT: $\vartheta : P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

(ii) E-CONTEXT: $t_1 = \vartheta' \quad \vartheta : P \rhd t_0; \rho \xrightarrow{\vartheta''} t_0'; \rho'$

Similar to (i).

Case T-REACTIVE: $t = r$

Cannot happen. The term t is already a value.

Case T-SIGNAL: $t = \mathsf{signal}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\, T_1$

There is no case by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived.

Case T-SOURCEVAR: $t = \mathsf{var}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Var}\, T_1$

There is only a single case by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-CONTEXT: $\vartheta : P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-NOW: $t = \mathsf{now}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\, T_1 \ \vee \ T = \mathsf{Var}\, T_1$

There is only a single case by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-CONTEXT: $\vartheta : P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-SET: $t = \mathsf{set}\, t_1 := t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Var}\, T_1$
$\Psi; \Gamma; \Gamma; P \vdash t_2 : T_1 \quad T = \mathsf{Unit}$

There are two cases by which $\vartheta_i : P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived: (i) E-CONTEXT if t_1 is not a value and (ii) E-CONTEXT if t_1 is a value and t_2 is not a value.

(i) E-CONTEXT: $\vartheta : P \rhd t_1; \rho \xrightarrow{\vartheta'} t_1'; \rho'$

We obtain $\vartheta_i = \vartheta$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

(ii) E-CONTEXT: $t_1 = v \quad \vartheta : P \rhd t_2; \rho \xrightarrow{\vartheta'} t_2'; \rho'$

Similar to (i). □

Theorem 13 (Static Placement on s-terms)**.** *If $\Psi; \Delta \vdash s$ and $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i : P_i' \rhd t_i; \rho_i \xrightarrow{\vartheta''} t_i'; \rho_i'$ holds $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$ and $P_i = P_i'$.*

Proof. Case analysis on the derivation of s.

Case T-PLACED: $s = \mathsf{placed}\, x{:}\, T\, \mathsf{on}\, P = t\, \mathsf{in}\, s_0 \quad \Psi; \Delta; \varnothing; P \vdash t : T$
$\Psi; \Delta, x{:}\, T\, \mathsf{on}\, P \vdash s_0$

There is only a single case by which $\vartheta_i{:}\, P_i' \rhd t_i; \rho \xrightarrow{\vartheta''} t_i'; \rho'$ for a subterm t_i can be derived:

E-PLACED: $\mathcal{I}^{[P]}{:}\, P \rhd t; \rho \xrightarrow{\vartheta'} t'; \rho'$

We obtain $\vartheta_i = \mathcal{I}^{[P]}$ and $P_i' = P_i = P$ and, thus, $\vartheta_i \subseteq \mathcal{I}^{[P_i']}$.

Case T-END: $s = \mathsf{unit}$

Cannot happen. The term s is already a value. □

Further, we prove that remote access is explicit, i.e., it is not possible to compose expressions placed on different peers without explicitly using asLocal:

Theorem 14 (Explicit Remote Access)**.** *If $\Psi; \Delta; \Gamma; P \vdash t : T$, then every subterm t_i of t with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ is either an explicitly accessed remote term (that is, t_r in one of* $\mathsf{asLocal}\, t_r{:}\, S$, $\mathsf{asLocal}\, t_r{:}\, S\, \mathsf{from}\, t_f$, $\mathsf{asLocal\ run}\, x{:}\, T = t_x\, \mathsf{in}\, t_r : S$ *or* $\mathsf{asLocal\ run}\, x{:}\, T = t_x\, \mathsf{in}\, t_r\, \mathsf{from}\, t_f : S$*) or $P = P_i$.*

Proof. By case analysis on the derivation of $t{:}\, T$.

Case T-VAR: $t = x$

The term t has no subterms.

Case T-APP: $t = t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_2 \to T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : T_2$
$T = T_1$

For both subterms t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ holds $P_i = P$.

Case T-ABS: $t = \lambda x{:}\, T_1.\, t \quad \Psi; \Delta; \Gamma; P \vdash \lambda x{:}\, T_1.\, t : T_1 \to T_2$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-UNIT: $t = \mathsf{unit}$

The term t has no subterms.

Case T-CONS: $t = \mathsf{cons}\, t_1\, t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad \Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{List}\, T_1$
$T = \mathsf{List}\, T_1$

For both subterms t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ holds $P_i = P$.

Case T-NIL: $t = \mathsf{nil\ of}\, T_1$

The term t has no subterms.

Case T-SOME: $t = \mathsf{some}\, t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Option}\, T_1$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-NONE: $t = \mathsf{none\ of}\ T_1$

The term t has no subterms.

Case T-PEER: $t = \vartheta$

The term t has no subterms.

Case T-ASLOCAL: $t = \mathsf{asLocal}\ t_1 \colon T_1\ \mathsf{on}\ P_1 \quad \Psi; \Delta; \varnothing; P_1 \vdash t_1 : T_1$
$P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

The term t has no subterms that are no explicitly accessed remote terms.

Case T-ASLOCALFROM: $t = \mathsf{asLocal}\ t_0 \colon T\ \mathsf{on}\ P_1\ \mathsf{from}\ t_1$
$\Psi; \Delta; \varnothing; P_1 \vdash t_0 : T_1 \quad P \leftrightarrow P_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Remote}\ P_1 \quad T = T_1$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-BLOCK: $t = \mathsf{asLocal\ run}\ x \colon T_0 = t_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x \colon T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1 \quad T = \Phi(P, P_1, T_1)$

For the subterm t_0 with $\Psi_0; \Delta_0; \Gamma_0; P_0 \vdash t_0 : T_0$ holds $P_0 = P$.

Case T-BLOCKFROM: $t = \mathsf{asLocal\ run}\ x \colon T_0 = t_0\ \mathsf{in}\ t_1 : T_1\ \mathsf{on}\ P_1\ \mathsf{from}\ t_2$
$\Psi; \Delta; \Gamma; P \vdash t_0 : T_0 \quad \Psi; \Delta; x \colon T_0; P_1 \vdash t_1 : T_1 \quad P \leftrightarrow P_1$
$\Psi; \Delta; \Gamma; P \vdash t_2 : \mathsf{Remote}\ P_1$

For the subterm t_0 with $\Psi_0; \Delta_0; \Gamma_0; P_0 \vdash t_0 : T_0$ holds $P_0 = P$ and for the subterm t_2 with $\Psi_2; \Delta_2; \Gamma_2; P_2 \vdash t_2 : T_2$ holds $P_2 = P$.

Case T-REACTIVE: $t = r$

The term t has no subterms.

Case T-SIGNAL: $t = \mathsf{signal}\ t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\ T_1$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-SOURCEVAR: $t = \mathsf{var}\ t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Var}\ T_1$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-NOW: $t = \mathsf{now}\ t_1 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : T_1 \quad T = \mathsf{Signal}\ T_1 \ \vee\ T = \mathsf{Var}\ T_1$

For the subterm t_1 with $\Psi_1; \Delta_1; \Gamma_1; P_1 \vdash t_1 : T_1$ holds $P_1 = P$.

Case T-SET: $t = \mathsf{set}\ t_1 := t_2 \quad \Psi; \Delta; \Gamma; P \vdash t_1 : \mathsf{Var}\ T_1$
$\Psi; \Gamma; \Gamma; P \vdash t_2 : T_1 \quad T = \mathsf{Unit}$

For both subterms t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ holds $P_i = P$. □